Part I
Aibo Programming

Introduction to the Aibo programming environment

Ricardo A. Téllez (r_tellez@ouroboros.org)

17th July 2005

Contents

1 Introduction and installation

Development with Aibo is possible using the OPEN-R SDE. The OPEN-R SDE is a set of tools released by Sony for the development of programs for Aibo. It is composed by the OPEN-R SDK, the R-Code SDK, the Aibo Remote Framework and the Aibo Motion Editor. This chapter describes how to set up a research environment with an Aibo robot for the use of those tools, but it does not include instructions about how to use the tools or how to program the robot. Those issues will be treated on the following chapters.

Basically, Aibo can be programmed by one of the following options: using the scripting language R-Code, programming in the C++ environment provided by OPEN-R, or using the Windows API provided by th Remote Framework. Usually, when using Aibo for complex control and scientific purposes only the OPEN-R option is used. Because of that, this document will focus on the creation of a development environment for the OPEN-R.

OPEN-R is composed of libraries and sample programs that allow a complete control over the Aibo hardware using a C++ environment. Even that this development kit is mainly targeted to a Linux platform, it is also possible to use it on Windows platforms by installing the Cygwin application. The usual procedure to work with Aibo is the following: a program is created on the PC computer using OPEN-R and C++. This program is then cross-compiled in the PC to target the Aibo platform. Then the resulting binaries are transferred from the PC to a memory stick. The memory stick is then inserted onto Aibo and executed by the robot.

To set up all the required environment, it will be necessary to install first the OPEN-R SDK on the PC. Then a installation on the PC of a memory stick reader/writer will also be necessary in order to transfer programs from the computer to the memory stick. Once the PC is able to use the memory stick reader/writer, it will be necessary to prepare a memory stick with the OPEN-R base system. The last step will be to configure the wireless network for communication between the PC and Aibo, and the whole programming environment will be ready for Aibo programming.

The used environment for the development of experiments with Aibo was composed of an Aibo ERS-220a equipped with a wireless LAN card, a computer running Windows XP and a wireless access point connected to the computer. A second environment was also proved to work perfectly, consisting of the new Aibo ERS-7 (which has a built in wireless card) and a Linux computer equipped with a wireless LAN card, working in Ad-hoc mode.

1.1 Installing the OPEN-R SDK on the PC

When starting to work with Aibo, the first thing to do is to visit the OPEN-R web page (http://openr.aibo.com) and download all the information and programs required. When accessing the page for the first time a registration will be required (a free one, though). After having the access code, the download section will give access to all the programs and documentation related to the SDE.

Download the following files for any operating system:

1. OPEN-R SDK
2. Documents English OPEN-R SDK for ERS-7/ERS-200 series
3. Sample programs
4. If you are working with a Windows machine also download:
5. Cygwin binaries
6. MIPS cross-development tools for Cygwin

Download the following files if you are using a Linux machine: gcc source files binutils source files newlib source files Shell script for building cross development tools

At this point, you should first decompress the OPEN-R SDK Documentation package and start reading the Installation Guide document. This document will guide you on how to install the OPEN-R SDK under Windows or Linux, and it is very straight forward. The installation of the OPEN-R SDK onto your PC will allow you to use the OPEN-R libraries to create your Aibo programs, and also to cross-compile those programs generating Aibo binaries.

Special mention is required for the installation of Remote Processing OPEN-R. This is an additional package of OPEN-R that allows a computer to run some of the

Aibo processes when the whole processing load is very high to be executed on Aibo itself. It allows the split of programs between Aibo and a computer host. To install this extra package just go to the */usr/local/OPEN_R_SDK/RP_OPEN_R/bin* directory and execute the setup-rp-openr script. Do not install the package if you do not plan to use it.

1.2 Installing the memory stick reader/writter

Installing the memory stick under Windows is very straight forward and has no problem at all. Just plug the USB into the computer and play. Windows XP comes already with the required drivers. For other versions of Windows, the driver may be required. Just use the CD provided with the reader/writer.

Installation under Linux is a little more complicated. You will need to have your kernel compiled with support for USB. Also you should include SCSI emulation and the usb-storage module. Once this has been done you can insert the memory stick reader/writer and mount it with the *mount* command. This process will not be described here since most modern Linux distributions perform automatically all those steps. If you require a list of the steps, please refer to the *Aibo quickstart manual* by R.Téllez available at www.ouroboros.org.

1.3 Installing the OPEN-R base system on the memory stick

At this point, you should first decompress the OPEN-R SDK Documentation package and start reading the Installation Guide document. This document will guide you on how to install the OPEN-R SDK under Windows or Linux, and it is very straight forward. The installation of the OPEN-R SDK onto your PC will allow you to use the OPEN-R libraries to create your Aibo programs, and also to cross-compile those programs generating Aibo binaries.

The preparation of the stick is a simple process: just copy some files provided by the OPEN-R SDK into an empty stick. Those files are the operating system of the robot. Three possible options of the operating system are possible: Basic, is a version of the operating system with all its capabilities but without wireless LAN environment; Wlan, with a wireless environment but no wireless console; Wconsole, with wireless environment and console (more about the console on section 8). On top of that, you must choose between an operating system with memory protection (memprot) where the operating system implements some control on memory accessing to prevent errors, or without it (nomemprot). You can find more information about it on chapter 4 of the *Programmer's Guide*.

Following your previous selection, copy the following directory into the root of the memory stick:

- For Basic environment with memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/BASIC/memprot/OPEN-R
  ```

- For Basic environment without memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/BASIC/nomemprot/OPEN-R
  ```

- For Wlan environment with memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/WLAN/memprot/OPEN-R
  ```

- For Wlan environment without memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/WLAN/nomemprot/OPEN-R
  ```

- For Wconsole environment with memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/WCONSOLE/memprot/OPEN-R
  ```

- For Wconsole environment without memory protection:

  ```
  /usr/local/OPEN_R_SDK/OPEN_R/MS_ERS7/WCONSOLE/nomemprot/OPEN-R
  ```

1.4 Setting up the wireless network

The wireless LAN network is not necessary in all cases, but it definitely will improve design speed by allowing direct transfer of new programs from the PC to the Aibo memory stick without having to take the stick out of Aibo. The wireless network can

also allow more specialized experiments of remote monitoring and control of the Aibo pet.

Depending on the parts you have purchased, you can establish a wireless LAN by using an access point or by using just a simple wireless card on a PC.

Note: In all the configuration cases, it has been taken the 192.168.10.x network by default. This can be easily changed by specifying another network in the configuration files explained. Just be consistent with all the configurations to use the same network.

1.4.1 Communication through an Access Point (AP)

In this case the PC is connected to an AP through an ethernet cable. The AP connects via wireless with Aibo. This is called managed mode. In this case, the PC communicates with Aibo via the AP which is acting like a gateway. Things to set up are the PC, the AP and the Aibo wireless card.

Configuration of the access point

You must set the following parameters on the AP. Configuration of the AP depends on the model purchased, but most of them allow web based configuration. Check the AP instructions for that.

ESSID: AIBONET
WEP key: AIBO2
Wireless channel: any between 1 and 11
IP address: 192.168.10.1
Netmask: 255.255.255.0
Operation mode: router

Configuration of the PC

You can configure the PC using the typical Control Panel under Windows or the *ifconfig* command under Linux.

IP address: 192.168.10.2 (or any available in the same range)
Netmask: 255.255.255.0
Gateway: 192.168.10.1

Configuration of the Aibo wireless card

Wireless configuration of Aibo requires to modify a special file included in one directory of the memory stick. You should skip this step until you understand how the memory stick works. For this, you should read the *Programmer's Guide* document and section 4 of this document.

The wireless configuration file is *OPEN-R\SYSTEM\CONF\WLANDFLT.TXT* found in the directory structure of the memory stick. If you edit this file you will see easily how to configure the robot for wireless communication. Please, use the following data to configure accordingly with the AP and the PC:

```
HOSTNAME=AIBO
ETHER_IP=192.168.10.100
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.10.1
ESSID=AIBONET
WEPENABLE=1
WEPKEY=AIBO2
APMODE=2 # this mode indicates auto-mode
CHANNEL=3
#DNS_SERVER_1=10.0.1.1
#DNS_SERVER_2=10.0.1.2
```

Hint: the auto-mode of Aibo allows it to boot the wireless network on managed or ad-hoc mode, depending on the detection or not of a nearby AP. This means that, when Aibo is booting, if it detects an AP, it will boot in managed mode. Otherwise, it will boot in ad-hoc mode. Other APMODEs are 0 for managed and 1 for ad-hoc.

1.4.2 Communication without access point

In this case, the PC communicates with Aibo directly by using a wireless LAN card. This is called ad-hoc mode. You should configure the PC wireless card and the Aibo wireless card.

Configuration of the PC

Use Control Panel under Windows, or *ifconfig/iwconfig* under Linux, to configure the wireless card with the following parameters:

ESSID: AIBONET
WEP Key: AIBO2
Wireless Channel: any between 1 and 11
IP address: 192.168.10.2 (or any available in the same range)
Netmask: 255.255.255.0
Communication mode: ad-hoc (also called Peer-to-Peer)

Configuration of the Aibo wireless card

Do the same as explained in 1.4.1

1.5 Brief OPEN-R SDK description

As shown in section 2, the SDK is mainly composed of a set of C++ libraries, some documentation files and a set of sample programs. Apart from reading the *Programmer's Guide* and the *Level2 Reference Guide*, sample programs are the best way to learn about OPEN-R. Sony doesn't have any specific OPEN-R programming tutorial but you will find a very good one at ENSTA (http://www.ensta.fr/~baillie/openr_tutorial.html) created by Jean-Christophe Baillie and François Serra. You can also find a similar tutorial by the author of this document at www.ouroboros.org.

The procedure to run a program on Aibo is as follows: having an empty memory stick, you first copy to it Aibo's operating system. Then you go to the directory in your PC that contains the program you want to run on Aibo and generate the binary files (cross-compilation). Next, you must modify the wireless LAN settings if necessary. Last, you copy the generated binaries into the memory stick, and plug it into Aibo.

The best way to start is to test this procedure with one of the sample programs. To compile and install any of the sample programs do always the same: go to the directory in your PC that contains the program (for example, I would like to compile the HelloWord sample program, and I uncompressed the sample programs under the */usr/local/sample_programs/ directory*) and type make install. This will create the binary files and place them into the */usr/local/sample_programs/HelloWord/MS/OPEN-R/MW/OBJS/* directory. Then transfer the contents of */usr/local/sample_programs/HelloWord/MS/OPEN-R* directory to the memory stick (this transfer process must not replace the existing directory, but to merge with the existing files. This means, do not replace one directory for the other, but overwrite common files). If you want to modify the wireless settings, do it in the */OPEN-R/SYSTEM/CONF/wlanconf.txt* file of the memory stick.

Any OPEN-R program on Aibo is composed of objects running concurrently. When you create a program for the robot, you must create all the required objects for your control program. Those objects will run independently one of the others, but will communicate between them through connections in order to behave coordinately. Most of the sample programs are already valid objects for your own programs, so it is a good idea to start creating your programs by modifying the existing sample ones.

Note: you can use Remote Processing OPEN-R to execute some of the objects on a PC and the rest of object on Aibo, sharing in this way the CPU load that has to support Aibo's processor (for more information on this see the *Programmer's Guide*).

1.6 Other Aibo programming environments

Apart from using OPEN-R to program the robot, you can use other higher level programming environment created by third parts. I would mainly comment the following two:

The **Tekkotsu** environment (http://www-2.cs.cmu.edu/~tekkotsu/index.html) Created at the Carnegie Mellon University, it provides a programming framework built on top of OPEN-R. It allows the programmer to easily access and control any of the sensors/actuators of the robot. It also provides powerful processing tools combined with other programs like Matlab for image processing. Everything is Open Source and you can download it for free. This environment also provides similar tools to those of Sony commercial packages like Navigator for free. Main environment is Linux but it can be run under Windows using Cygwin (some bugs reported).

The **URBI** scripting language (http://www.urbiforge.com/index.html) that provides remote control of the robot. By using a client-server architecture you can command your Aibo from a remote computer through wireless LAN. It also contains a C++ library to allow the control inside your own C++ programs.

The **Pyro** environment (http://emergent.brynmawr.edu/pyro/) is another programming environment that allow the control of Aibo by using the Python programming language. It is built on top of Tekkotsu.

Yart/RCodePlus (http://www.aibohack.com/rcode/index.html) This is an environment built on top of the Rcode of Sony, that allows the easy creation of behaviors on Aibo by using a scripting language that boosts the power of the original Rcode. It is easy to use and program but functionality is limited by its easiness. However, it contains some interesting tools added, like remote control of Aibo, wireless consoles, etz.

1.7 Tips and tricks

Description of some useful features

1.7.1 Printing Aibo messages to the wireless console

If you install in the memory stick the operating system version that has a wireless console (see Programer's Guide), then you can telnet Aibo from a PC for monitoring and debugging purposes. Connection is accomplished by telnetting at the 59000 port of Aibo:

```
> telnet Aibo_IP 59000
```

You can include debugging messages in your Aibo code to appear in the telnet console. The allowed calls for messages are *OSYSPRINT()* and *OSYSDEBUG()*. The first one will print always the message indicated. The second one will only print on console the message if the debug flag was set during compilation.

1.7.2 Using the FTP server to transfer a program to Aibo without using the memory stick reader/writer

You can avoid the extraction and insertion process of the memory stick anytime you want to change Aibo's program, by using an FTP server. The FTP server is an Aibo program provided by Sony on the samples directory, called TinyFTPD, that allows direct transfer of any new program from the PC to the Aibo memory stick, when the stick is in Aibo. It uses FTP protocol, so any typical FTP client can be used on the PC to transfer the files.

To use that program, you must first compile it and install it onto the memory stick. So, you must go to the TinyFTPD directory (samples/TinyFTPD/) and do make install. Then you will have to copy the generated object *samples/TinyFTPD/MS/OPEN-R/MW/OBJS/TINYFTPD.BIN* to the */OPEN_R/MW/OBJS/* directory of the memory stick. Next, you must add a line containing */OPEN-R/MW/OBJS/TINYFTPD.BIN* to the file */OPEN-R/MW/CONF/OBJECT.CFG* (of the memory stick). Last, add the file *samples/TinyFTPD/MS/OPEN-R/MW/CONF/PASSWD* to the */OPEN-R/MW/CONF/* directory of the memory stick.

When Aibo is running this new program, it will be possible to access Aibo's memory stick using any FTP client on the PC. Just type ftp AIBO_IP. The server will answer requesting a username and a password, where you can answer anonymous on both cases.

Once connected you can use the typical FTP commands to transfer, delete and copy new files to the Aibo memory stick. If you upload a new program to Aibo and what to reboot it, just type the following FTP command: quote rebt.

Hint: the auto-mode of Aibo allows it to boot the wireless network on managed or ad-hoc mode, depending on the detection or not of a nearby AP. This means that, when Aibo is booting, if it detects an AP, it will boot in managed mode. Otherwise, it will boot in ad-hoc mode.

1.7.3 Hints

- To have a good overview of Aibo's capabilities, including physical limits, return values of sensors and output devices, consult the document *Model Information for ERS-xxx.*
- Use *OSYSPRINT* for debugging on a wireless console environment.
- Execute *make clean* after any modification in any of the *.h files, since it can clear some untraceable segmentation fault errors.
- It has to be taken into account that Aibo's MIPS processor is little-endian when transferring data to/from systems working in big-endian mode.

1.8 Interesting links

Apart from the well known page of the OPEN-R SDK (http://openr.aibo.com) there are just a few good web pages related to the insights of Aibo. Nevertheless hundreds of pages can be found with no interest for research. Only the first type of pages are listed here:

- OPEN-R SDK Tutorial at ENSTA: a very good tutorial about OPEN-R available on the network http://www.ensta.fr/~baillie/openr_tutorial.html
- OPEN-R Essentials: another tutorial about OPEN-R based on the previous one http://www.ouroboros.org/
- R-Code SDK Tutorial: the first tutorial on the net about R-Code programming http://www.ouroboros.org/
- Sony OPEN-R SDK school: 6 powerpoint tutorials describing some of the sample programs http://openr.aibo.com/ (OPEN-R SDK University section of the registered area)
- Aibo Hack: a web site with lots of hacks for Aibo http://www.aibohack.com
- Aibo life bot house: an interesting forum where to learn and ask about Aibo, including technical details http://www.aibo-life.org/
- Dogsbody net: good extension programs for Aibo http://www.dogsbodynet.com/

2 OPEN-R definitions

This section describes the components of OPEN-R programs and how do they relate in order to form a complete control program for Aibo. We will use several sample programs provided by Sony as example of the explanations.

2.1 Objects

An OPEN-R program consists of a set of **objects** that are executed concurrently on the robot, plus a set of **configuration files** that specify how those objects must interact. The objects are cross-compiled in the host machine producing Aibo code. This code is then transferred to a memory stick that is inserted on the Aibo robot and executed on it.

An OPEN-R object could be defined as a **process** running on Aibo. An OPEN-R program is just a set of those objects running concurrently (in parallel). OPEN-R objects communicate with each other in order to coordinate. Thus, a program to control Aibo will consist of a set of OPEN-R objects, each object performing its own job but coordinating with the other objects by using message passing. An OPEN-R object corresponds to one executable file for Aibo created at compile-time. When Aibo boots, all the compiled objects are loaded into memory and started as concurrent processes. Their message interchange will determinate the flow of action.

The behaviour of every object is described as the transition between its **internal states**. Each object is based on its **present state** and the **transitions** that lead to other states. This means that an object **will be always** on a state. **The design of an object is the design of its required states, transitions and functions to apply when going from one state to another**.

Summarising, to design an object, it must be specified its internal states. An object can be only in one state at a time. Objects change their state by means of transitions. Transitions are activated by the reception of messages, and can have several paths going to several states. Only the path that satisfies the condition will be taken. As you will see, conditions must be exclusive in order to do not allow the object be in two different states at the same time.

2.2 The base class

The base class is the C++ class that will represent an object in OPEN-R. Each object created by the programmer will inherit from the base class and will be represented by only one of those class. The base class name is *OObject* and contains 4 virtual functions:

- *OStatus DoInit (const OSystemEvent& event)*
- *OStatus DoStart (const OSystemEvent& event)*
- *OStatus DoStop (const OSystemEvent& event)*
- *OStatus DoDestroy (const OSystemEvent& event)*

Those functions perform some basic functions that will guide the start-up of the object and its shutdown, and they must be implemented in the code of the programmer's object (but most of the code for those functions will be handled by some macros provided by OPEN-R, as you will see below).

DoInit and *DoStart* are initialisation functions. They are called automatically by this order by OPEN-R once the object has been loaded into memory. When the object is in memory, then its *DoInit* function is called to initialise the object. When finished, the *DoStart* function is called to perform some stating actions.

DoStop and *DoDestroy* are called when OPEN-R is shutting down Aibo. They perform cleaning and object closing tasks.

When creating an OPEN-R object, it inherits from the base class. This means that the new object created has to redefine those virtual functions inside its object construction code. The constructor of the object must also indicate its starting state (usually, it starts in IDLE state), and how many other states do exist for that object.

2.2.1 The HelloWord example

The provided Helloword example is very useful to understand the behavior of the virtual functions and the flow of execution. The code of the HelloWord.cc file is listed below:

```
#include <OPENR/OSyslog.h>
#include "HelloWorld.h"
HelloWorld::HelloWorld ()
{
  OSYSDEBUG(("HelloWorld::HelloWorld()\n"));
}
OStatus HelloWorld::DoInit(const OSystemEvent& event)
{
  OSYSDEBUG(("HelloWorld::DoInit()\n"));
  return oSUCCESS;
}
OStatus HelloWorld::DoStart(const OSystemEvent& event)
{
  OSYSDEBUG(("HelloWorld::DoStart()\n"));
  OSYSPRINT(("!!! Hello World !!!\n"));
  return oSUCCESS;
}
OStatus HelloWorld::DoStop(const OSystemEvent& event)
{
  OSYSDEBUG(("HelloWorld::DoStop()\n"));
  OSYSLOG1((osyslogERROR, "Bye Bye ..."));
  return oSUCCESS;
}
OStatus HelloWorld::DoDestroy(const OSystemEvent& event)
{
```

```
  return oSUCCESS;
}
```

2.2.2 Compilation, instalation and execution

To compile and execute the HelloWord sample program the following files are relevant: *Makefile*, to compile the object; *HelloWord.cc* and *HelloWord.h*, that contain the code of the object; *helloWorld.ocf*, that contains the object configuration; and *object.cfg* that indicates which objects will be executed.

- The *Makefile* file is just a typical compilation file that indicates the requirements for compilation and generation of the executables on Aibo. Study it to understand the dependencies. It will not be explained here since it is not related to OPEN-R but to typical C++ compilation.
- The *helloWord.ocf* file is required for every object you design. It specifies the configuration of the object and has the following format:

  ```
  object OBJECT_NAME STACK_SIZE HEAP_SIZE
         SCHED_PRIORITY CACHE TLB MODE
  ```

  - OBJECT_NAME is the name of the object
  - STACK_SIZE is the size of the stack in bytes. The stack size is fixed during the execution to this value.
  - HEAP_SIZE is the size of shared memory that will be reserved. Shared memory is used to message passing between object and is very important.
  - SCHED_PRIORITY indicates the priority of the object by means of a byte. the first four bits indicate the type of planification and the 4 minor bits indicate the ratio of execution of objects with the same type of planification.
  - CACHE is a parameter with two possible values: 'cache' or 'nocache'. It indicates if it has to be used the processors cache (recommended)
  - TLB is a parameter with two possible values: 'tbl' touse memory on the virtual address space, or 'notbl' to use the physical address space.
  - MODE can have two possible values: 'kernel' to execute the object in kernel mode, or 'user' to execute the object in user mode.
- The *object.cfg* file, located on another directory, *MS/OPEN-R/CONF/object.cfg*, contains a list of all the objects that are going to be executed on Aibo.

To compile the object go to the superior HelloWord directory and type:

```
make clean
make
make install
```

This will compile all required files and put them under the MS directory. To install the compiled program into Aibo, just **merge** the content of *MS/OPEN-R* with the *OPEN-R* directory of the memory stick (OPEN-R base system should be instaled already on the memory stick, following the instructions of section 1.3.

The flow of execution is the following: first the HelloWord object is created. Then the DoInit procedure is executed, and when finished *DoStart* is executed. You can see the '!!! Hello Word !!!' message on the telnet connection (see section 1.7 about how to connect to the console). When *DoStart* is finished, the OPEN-R program waits on an **undefined** state for an event. Since no events have been defined yet (see on next subsection) only the shutdown event can take the program out of that state. So pushing the *Power On* button of the robot will start the execution of *DoStop* and after, *DoDestroy*.

OSYSPRINT, *OSYSDEBUG* and *OSYSLOG1* are macros provided by OPEN-R to allow the user to print messages on the wireless console. The main difference between then is, while *OSYSPRINT* allows the printing of any message, *OSYSDEBUG* prints on the console only when the *-DOPENR_DEBUG* flag has been activated during compilation. *OSYSLOG* prints the message selected indicating also the type of error and its priority.

2.3 Communication between objects

Objects communicate with each other by message passing. In every inter-object communication the sender of the message is called the **subject** and the receiver is called the **observer**. An object can act as subject in one situation but as observer in another one. It just depends on the situation.

The communication channel between two objects is unidirectional (each channel has a fixed subject and a fixed observer). It means that, if bidirectional communication is required between two objects, then two different channels will be required. Also, an object can have different subjects and be the observer of several objects, but a communication channel will be required for each one. Messages go out of the object and come in through **gates**. There is a gate in the object for each communication channel.

Since objects are single threaded, it means that they can only process one single message at a time. For this purpose, a message queue is implemented in every object, where messages wait their turn to be processed.

The main flow execution of an object is the following:

1. The object is initialised. Then it sends an AR message to all its subjects.
2. The object waits on a determined state the arrival of a message coming from one of its subjects
3. Once the message arrives, the method associated to that message is activated. Within that method, the message is processed
4. Once finished the processing, the object sends an AR message to the subject indicating that is already ready to receive new messages
5. It returns to point 2.

The steps number one and two are realised by the *DoInit* and *DoStart* virtual functions. The programmer must create those functions in order to initialise the objects. In this case, in oposition to the case of the HelloWord example, the objects need to communicate with other objects. Because of that, the initialisation procedure is a little more complex than in the HelloWord example. All the initialisation required for communication is handled by some primitives provided by OPEN-R. You should use them in your code when the objects require communication with other objects.

- Example: initialisation of the *SampleSubject* object of the *ObjectComm* sample program

```
OStatus SampleSubject::DoInit(const OSystemEvent& event) {
    NEW_ALL_SUBJECT_AND_OBSERVER;
    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;
    return oSUCCESS;
}
OStatus SampleSubject::DoStart(const OSystemEvent& event) {
    ENABLE_ALL_SUBJECT;
    ASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}
```

In the same way, when the objects are switched off at shutdown, they must close all communications with other objects. This is also achieved by using some OPEN-R primitives in *DoStop* and *DoDestroy*.

- Example: shutdown of the *SampleSubject* object

```
OStatus SampleSubject::DoStop(const OSystemEvent& event) {
    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}
OStatus SampleSubject::DoDestroy(const OSystemEvent& event)
{
    DELETE_ALL_SUBJECT_AND_OBSERVER;
    return oSUCCESS;
}
```

AR messages are special messages defined within OPEN-R. These messages, whose complete name is ASSERT_READY, are part of the synchronisation protocol used to let the observer notify the subject that it is ready to receive new messages. This is usually done after the last received message has been processed.

Messages sent between objects can be of any C++ primary type, or an array, an structure, a class or a pointer. However, their length is limited to the size of *OCommandVectorData* (more on this on section 3). For this reason, it is a common technique to send pointers to common shared memory as messages, and put on that shared memory the information that wants to be sent (more on section 4).

In order to be able to communicate with other objects, the constructor of the class must define two arrays, one that will contain a list of the subjects (called *subject*) and another for the list of the observers (called *observer*). The programmer has just to care about defining those arrays and use them, but all the construction and filling process is done automatically by the virtual functions described above (see below). Second, OPEN-R also defines a set of indexes for easy access to the array. Those indexes are also generated at compile time based on the information provided in the *stub.cfg* file. The *stub.cfg* file contains information about the gates (also called services) of the object (more about this file in section 2.3.1). Since every gate has a name, the array indexes are created in the following way: every index is identified by a name. The index name is formed by the concatenation of the type of the service (*sbj* for subject and *obs* for observer) and the name of the gate. For example, for the *SampleSubject* object, there is an outgoing gate called *SendString* that connects the object with the *SampleObserver* object. If it were necessary to access that subject, the sentence would be *subject[sbjSendString]*.

2.3.1 The stub.cfg config file

The *stub.cfg* file is the file that describes how the object connects with other objects. It describes the gates, including all the information to completely describe a gate. Every object has its own *stub.cfg* file, an the information of this file will be used by the compiler when building the binaries. The file must be placed in the same directory where the C++ object program resides.

Here there is an example of *stub.cfg* file for the *SampleSubject* object:

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

the first line describes the name of the object. Second and third line describe the number of subjects and observers the object has. Lines starting by *Service* are the ones that describe the information of the gates . For example, in line

```
Service : "SampleSubject.SendString.char.S", null, Ready()
```

SampleSubject is the name of the current object. *SendString* is the name of the gate the message will go through. *char* is the type of message been interchanged, and S means that the gate is outgoing (the current object is a subject). If the object was a observer then the last field should say S. The *null* is the name of a function to be executed when

a connection result is received. Most of the cases is null. The *Ready* is the name of the function to be executed when an AR (for subjects) or a message (for observers) is received through this gate.

- Example: the *Ready* routine for *SampleSubject*. It is activated when the *SampleObserver* object is ready for receiving a message from *SampleSubject*

```
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++;
}
```

2.3.2 The connect.cfg config file

The *connect.cfg* file is a configuration file that specifies how objects interconnect to each other. This is a **unique file per program** and it must be placed in the *OPEN-R/MW/CONF/* directory of the memory stick.

Lines starting by # are comments, and usually describe the connection between objects being described below. Each no-comment line specifies one connection, starting by the name of the subject and ending by the name of the observer.

- Example: the *connect.cfg* file for the *ObjectComm* sample program

```
SampleSubject.SendString.char.S SampleObserver.ReceiveString.char.O
```

specifies how the *SampleSubject* object connects with the *SampleObserver* object, acting the first as the subject and the second as the observer. For bi-directional connections, the configuration file must contain two lines, one for each direction of the connection. The definition of each part of the line follows the same specification as for the *stub.cfg* file described before. Special attention must be paid to assure that in one line, the messages exchanged between subject and observer have the same type.

2.4 System objects

Together with the objects created by the designer of the program, there exist other objects already created by OPEN-R that allow the access to the hardware of the robot. It means that access to sensors or actuators will be done through an OPEN-R object provided by the system. Those objects are:

- *OVirtualRobotComm*: it interfaces with the robot joints, sensors, LEDs and camera
- *OVirtualRobotAudioComm*: it interfaces with the robot audio devices for recording of playing sounds

The use of those objects in the programs is the same as the use of programmer defined objects. Some input and output points (the gates) in those objects are already defined to send and receive messages from/to them.

3 Accessing sensors using OPEN-R

This chapter describes how to obtain values sensed by Aibo's sensors and how to send commands to its actuators (joints and LEDs). All this process will be achieved by interacting with the virtual object *OVirtualRobotComm* provided by OPEN-R. Special treatment is required for the sound (which uses *OVirtualRobotSound* object), but this will not be treated in this tutorial.

3.1 Sensors and actuators primitives

In Aibo, every sensor and actuator are also called primitives. Each primitive has its own primitive locator to access to it. The **primitive locator** is like the path or the address you must follow to reach the desired sensor or actuator. A list of the available primitive locators is provided by Sony on its *Model Information* documentation. Primitive locators look like this:

- PRM:/r1/c1/c2/c3-Joint2:13 <– primitive locator for the HEAD TILT2 motor of ERS-7

The primitive locator cannot be used directly to access the sensor or actuator. Instead of it, the **primitive ID** must be used. The primitive ID is a number that identifies the sensor/actuator within the data structures returned by *OVirtualRobotComm* (see section 3.3). Because of that, a translation from the primitive locator to the primitive ID must be performed. Even that Sony also provides a conversion table on its documentation, it is recommended not to use it, since it may change for future robot models. Instead of that, an on-line conversion is recommended using the *OPENR::OpenPrimitive* function provided by OPEN-R. This function also performs some initialisation job, so it must be called once in the program before accessing the sensor/actuator. Usually such conversion is done once during initialisation phase and the results stored in an array for later use.

- Example: SensorObserver7 sample program translation from primitive locator to primitive ID

```
void SensorObserver7::InitERS7SensorIndex
(OSensorFrameVectorData* sensorVec)
{
    OStatus result;
    OPrimitiveID sensorID;
    for (int i = 0; i < NUM_ERS7_SENSORS; i++)
    {
       result = OPENR::OpenPrimitive(ERS7_SENSOR_LOCATOR[i],
                                     &sensorID);
       if (result != oSUCCESS)
       {
            OSYSLOG1((osyslogERROR, "%s : %s %d",
                  "SensorObserver7::InitERS7SensorIndex()",
```

```
                    "OPENR::OpenPrimitive() FAILED",
                    result));
            continue;
        }
      [...]
    }
}
```

3.2 Frames

Aibo's time is divided in frames. A frame is the unit of time and represents 8 ms. Information from sensors is retrieved by blocks of n frames (usually, 4 frames), which are contiguous in time. Commands to effectors are also sent in blocks of frames, it is, when you send a command to an effector, you must provide the commands for the next n frames of time.

3.3 Reading sensor values

In order to read a value from a sensor, it will be necessary to obtain messages sent by the *OVirtualRobotComm* through its gate named *Sensor*. Through that gate, *OVirtualRobotComm* sends a message of type *OSensorFrameVectorData* that contains all information related to the robot sensors. Therefore, the first step to catch that message would be to refer to the subject in the *connect.cfg* file using the following line:

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S
your_observer.your_observer_gate.OSensorFrameVectorData.O
```

An additional line must be also added to the *stub.cfg* file of the observer, indicating the name of the routine that will handle the message. The contents of the message is treated in the next section.

3.3.1 Description of the OSensorFrameVectorData message

The *OSensorFrameVectorData* is a data structure that accommodates all the required information to obtain Aibo's sensors states. It is formed by three main groups of data:

- *vectorInfo*: It is another structure of data type *ODataVectorInfo*. It contains the *numData* and *maxNumData* values. *numData* contains the number of sensors whose values are included in the structure (have been sensed).
- *OSensorFrameInfo*: it is an array of data structures. The structure contains information that identifies the sensor.
- *OSensorFrameData*: it is another array of data structures of the same length as *OSensorFrameInfo*. Each *OSensorFrameInfo*, has its correspondent *OSensorFrameData*. While *OSensorFrameInfo* identifies the sensor that is being read, *OSensorFrameData* contains its values sensed. Values are specified in a structure to allow the allocation of different frames.

Each cell of *OSensorFrameInfo* has a correspondent cell in *OSensorFrameData*. It means that the information in frame n of *OSensorFrameInfo* is related to the info of frame n in *OSensorFrameData*. Those two have the information related to a given sensor during the last frames.

vectorInfo

vectorInfo contains two values: *numData* and *maxNumData*. The two arrays *OSensorFrameInfo* and *OSensorFrameData* have an allocated memory size equal to *maxNumData*, but their actual size is that indicated by *numData*, it is, only *numData* sensors will have their values put in the *OSensorFrameData* array.

OSensorFrameInfo

Data from this array is available by using the *GetInfo (int index)* function. By using that function, the user obtains a *OSensorFrameInfo* structure that contains the following information: *type* is a variable of class *ODataType* and contains the type of sensor been access; *primitiveID* is a variable of type *OPrimitiveID* and contains the *primitive ID* of the sensor been access; *frameNumber* is a longword containing a tag number that identifies the first frame on its associated *OSensorFrameData* cell; *numFrames* indicates the number of frames that are valid in the associated *OSensorFrameData* cell.

OSensorFrameData

Data from this array is available by using the *GetData (int index)* function. By using that function, the user obtains a group of frames containing *OSensorValue* structures. These structures are generic data ones for sensor values. This means that every sensor will send their values in a subclass of *OSensorValue*. Usually a cast to the correct subclass is performed when retrieving a sensor value.

3.3.2 Accessing a sensor value

The main steps to access a sensor value are the following:

1. A *OSensorFrameVectorData* is received from *OVirtualRobotComm*
2. Get the primitive ID corresponding to the sensor to access. A complete list of the desired sensor IDs is usually created during initialisation by the programmer (see section 3.1), so a retrieve of the list will do it.
3. Compare the obtained primitive ID with the ones that are in the primitiveID field of the *OSensorFrameInfo* array. Once matched, the index within that array provides the index within the *OSensorFrameData* array that contains the sensor value. You can store this index in a user array since it will not change during the execution of the OPEN-R program (see the example below).
4. Use the index obtained to access the sensor value in *OSensorFrameData*.

5. Process the data received and take actions if required.
6. Do not forget to send an AR message after processing the data, to indicate that you can now receive another sensor message if available.

- Example: creating the correspondence table between primitive ID and index within OSensorFrameData

```
[...]
        for (int j = 0; j < sensorVec->vectorInfo.numData; j++)
        {
            OSensorFrameInfo* info = sensorVec->GetInfo(j);
            if (info->primitiveID == sensorID)
            {
                ers7idx[i] = j;
                OSYSPRINT(("[%2d] %s\n",
                           ers7idx[i],
                          ERS7_SENSOR_LOCATOR[i]));
                break;
            }
        }
[...]
```

- Example: accessing values

```
void SensorObserver7::NotifyERS7(const ONotifyEvent& event) {
    OSensorFrameVectorData* sensorVec =
                         (OSensorFrameVectorData*)event.Data(0);
    if (initSensorIndex == false) {
        InitERS7SensorIndex(sensorVec);
        initSensorIndex = true;
    }
    OSYSPRINT(("ERS-7 numData %d frameNumber %d\n",
               sensorVec->vectorInfo.numData,
               sensorVec->info[0].frameNumber));
    PrintERS7Sensor(sensorVec);
    WaitReturnKey();
    observer[event.ObsIndex()]->AssertReady();
}
void SensorObserver7::PrintERS7Sensor(OSensorFrameVectorData* sensorVec)
{
    PrintSeparator();
    //     // BODY     //
    OSYSPRINT(("ACC X      | "));
    PrintSensorValue(sensorVec, ers7idx[ACC_X]);
[...]
```

```
}
void SensorObserver7::PrintSensorValue(OSensorFrameVectorData* sensorVec,
                                       int index)
{
    if (index == -1) {
        OSYSPRINT(("[%d] INVALID INDEX\n", index));
        PrintSeparator();
        return;
    }
    OSensorFrameData* data = sensorVec->GetData(index);
    OSYSPRINT(("[%2d] val    %d %d %d %d\n",
               index,
               data->frame[0].value, data->frame[1].value,
               data->frame[2].value, data->frame[3].value));
    OSYSPRINT("           | "));
    OSYSPRINT("     sig    %d %d %d %d\n",
               data->frame[0].signal, data->frame[1].signal,
               data->frame[2].signal, data->frame[3].signal));
    PrintSeparator();
}
```

4 Accessing actuators using OPEN-R

This section describes all the required steps to send a command to an effector. We will use the *MovinHead7* example to show the working of the theory.

4.1 Configuration of the connect.cfg file

Sending commands to the Aibo actuators (these are motors and LEDs), requires to send a message of type *OCommandVectorData* to the *OVirtualRobotComm* object via its incoming gate, named *Effector* (remember that the outgoing gate of this object, from where values sensed come, was called *Sensor*). To be able to connect to the virtual object, the following line must be added to the *connect.cfg* file:

```
your_object.your_object_gate.OCommandVectorData.S
OVirtualRobotComm.Effector.OCommandVectorData.O
```

In the *MovinHead7* example, we can find the following line inside its *connect.cfg* file, indicating the connection between the *MovingHead7* object and the *OVirtualRobotComm*:

```
MovingHead7.Move.OCommandVectorData.S
OVirtualRobotComm.Effector.OCommandVectorData.O
```

4.2 Description of the OCommandVectorData message

The *OCommandVectorData* message is a structure very similar to *OSensorFrameVectorData* (used to read sensor values). It also contains three members, which are a structure called *vectorInfo*, and two arrays called *OCommandInfo* and *OCommandData*. Each cell of *OCommandInfo* has a corresponding cell in the *OCommandData* array.

vectorInfo

It is a structure of type *ODataVectorInfo*. It contains two elements called *numData* and *maxNumData*. The two arrays *OCommandInfo* and *OCommandData* have an allocated memory size equal to *maxNumData*, but their actual size is that indicated by *numData*, it is, only *numData* actuators will be available for manipulation in the *OCommandData* array.

OCommandInfo

This array is accessible by using the *GetInfo(int index)* function. Each cell in the *OCommandInfo* array has three main elements: *type* that describes the type of effector, *primitiveID* describing the effector primitive ID and *numFrames*, describing the number of frames passed in the command (since commands can be grouped to be sent for several frames).

OCommandData

It is a structure composed of an array of *OCommandValue*. *OCommandValue* is the general type for commands for effectors. Subclasses exists for each effector, and they must be checked at the *Model Information* documentation. To access the *OCommandData* structure use the *GetData(int index)* function. Each cell of *OCommandData* contains the commands for the next *numFrames* for a specific effector.

4.3 Sending a command to an effector

The list of steps to send a command to an effector :

Initialising the system

Initialisation is required in order to move Aibo's joints or to send commands to the LEDs. Depending on the effector we want to act, initialisation will require more or less work. Initialisation of the joints can be performed in any OPEN-R object. Only one initialisation is required, usually done at the initialisation step of an object (*DoInit* procedure). This initialisation includes setting the power of the joints on by using an OPEN-R primitive:

```
OPENR::SetMotorPower(opowerON);
```

Get primitive IDs

As happened with sensors, each effector has its own primitive ID. It is required to obtain the primitive ID of an effector in order to identify its index inside the *OCommandInfo* and *OCommandata* arrays. The steps to follow are the same as for sensors. First, the primitive locator is found on Model Information documentation. Then the *OPENR::OpenPrimitive()* function is used to make the conversion from locator to ID. This result usually is stored in an array for further use. In the *MovingHead7* example, the complete direction of the primitives to use is stored in the *MovingHead7.h* file:

```
static const char* const JOINT_LOCATOR[] = {
   "PRM:/r1/c1-Joint2:11", // TILT1
   "PRM:/r1/c1/c2-Joint2:12", // PAN
   "PRM:/r1/c1/c2/c3-Joint2:13" // TILT2
};
```

and the conversion from primitive direction to primitive ID is performed by the following code inside *MovingHead7.cc*:

```
void MovingHead7::OpenPrimitives()
{
   for (int i = 0; i < NUM_JOINTS; i++)
   {

OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i], &jointID[i]);
```

```
      if (result != oSUCCESS)
      {
        OSYSLOG1((osyslogERROR, "%s : %s %d",
                                  "MovingHead7::OpenPrimitives()",
                                 "OPENR::OpenPrimitive() FAILED", result));
      }
    }
  }
```

Set joint gains (for joints)

The joint motor control in Aibo is implemented by using a PID. In order to control those motors, their PID gains and shifts values must be set. The values to set the joints are specified by Sony on its *Model Information* documentation. Other PID values can be set but this is not recommended since they may damage the joints. Note that the PID values are different for each joint motor.

The first step is to enable the joint gains. To do this, just call the *OPENR::EnableJointGain()* function. Then you can set the gains by calling the *OPENR::SetJointGain()* function with the appropriate values.

- Example: values for the head joints, in the *MovingHead7.h* file

```
static const double TILT1_ZERO_POS     = 0.0;
static const double PAN_ZERO_POS       = 0.0;
static const double TILT2_ZERO_POS     = 0.0;
static const double SWING_AMPLITUDE    = 80.0;

static const word   TILT1_PGAIN        = 0x000a;
static const word   TILT1_IGAIN        = 0x0004;
static const word   TILT1_DGAIN        = 0x0002;

static const word   PAN_PGAIN          = 0x0008;
static const word   PAN_IGAIN          = 0x0002;
static const word   PAN_DGAIN          = 0x0004;

static const word   TILT2_PGAIN        = 0x0008;
static const word   TILT2_IGAIN        = 0x0004;
static const word   TILT2_DGAIN        = 0x0002;

static const word   PSHIFT             = 0x000e;
static const word   ISHIFT             = 0x0002;
static const word   DSHIFT             = 0x000f;
```

- Example: the code to set the gains in the head joinst (*MovingHead7.cc*):

```
void MovingHead7::SetJointGain() {
```

```
    OPENR::EnableJointGain(jointID[TILT1_INDEX]);
    OPENR::SetJointGain(jointID[TILT1_INDEX],
                        TILT1_PGAIN,
                        TILT1_IGAIN,
                        TILT1_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT);
    OPENR::EnableJointGain(jointID[PAN_INDEX]);
    OPENR::SetJointGain(jointID[PAN_INDEX],
                        PAN_PGAIN,
                        PAN_IGAIN,
                        PAN_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT);
    OPENR::EnableJointGain(jointID[TILT2_INDEX]);
    OPENR::SetJointGain(jointID[TILT2_INDEX],
                        TILT2_PGAIN,
                        TILT2_IGAIN,
                        TILT2_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT); }
```

Calibrate the joints (for joints)

It happens sometimes that the position read by the sensors and the real position of the joint differs in some small quantities. For this reason, before moving the joints it is usually performed a calibration step. It consists of reading the actual value of the joint and then setting the joint to the value sensed. Reading a joint value can be performed with the *OPENR::GetJointValue()* function. Then, the joint must be set to the read value by using a user defined function (not provided by OPENR).

- Example: the code for calibration in *MovingHead7.cc*

```
MovingResult MovingHead7::AdjustDiffJointValue() {
    OJointValue current[NUM_JOINTS];
    for (int i = 0; i < NUM_JOINTS; i++) {
        OPENR::GetJointValue(jointID[i], &current[i]);
          SetJointValue(region[0], i,
                        degrees(current[i].value/1000000.0),
                        degrees(current[i].value/1000000.0));
    }
    subject[sbjMove]->SetData(region[0]);
    subject[sbjMove]->NotifyObservers();
    return MOVING_FINISH;
}
```

Select a free shared memory region

Commands for effectors are not directly send. Instead of that, a buffer method is implemented in order to avoid two possible problems: first, messages maximum size may be smaller than the message being actually sent. Then, instead of sending a command structure (of type *OCommandVectorData*), objects send a pointer to a command structure situated in the shared memory. The shared memory is a place of Aibo's memory where all objects can write and read. Thus, it is used to interchange information between objects. Second, by using this method, a group of buffers can be set. Buffers would act as a place where commands are stored for retrieval when the *OVirtualRobotComm* is ready. By using those buffers it is possible to send commands to the *OVirtualRobotComm* without paying attention if its is ready or not. Commands are then just stored in the buffers waiting for the virtual object. This method brings smoothness and higher reactivity to the robot since every command is processed as quick as possible between gaps in the middle. Usually, two buffers are allocated. Buffers are created by the programmer.

To access the shared memory region, OPEN-R provides the *RCRegion* class. To send commands to joints OPEN-R provides a function (*OPENR::NewCommandVectorData()*) to allocate the memory and to hold a reference counter. This counter is used by the system to avoid region overwriting. The OPEN-R command creates a *OCommandVectorData* in shared memory and an ID for that memory. It needs three arguments:

- *size_t numCommands*, that contains the number of cells in the *OCommandData* array, one for each actuator wanted to command
- *MemoryRegionID* memID*, that will have the ID of the memory allocated
- *OCommandVectorData** baseAddr*, that is the pointer to the memory region

Once called this function, then the *RCRegion* class must be instantiated. The class constructor is *RCRegion (MemoryRegionID memID, size_r offset, void* baseAddr, size_t size)* where *memID* is the *memRegionID* of the *ODataVectorInfo*, *offset* is the offset of *ODataVectorInfo*, *baseAddr* is the pointer returned by *OPENR::NewCommandVectorData()* and *size* is the total size of *ODataVectorInfo*.

- Example:

```
void MovingHead7::NewCommandVectorData()
{
   OStatus result;
   MemoryRegionID cmdVecDataID;
   OCommandVectorData* cmdVecData;
   OCommandInfo* info;
   for (int i = 0; i < NUM_COMMAND_VECTOR; i++)
   {
      result = OPENR::NewCommandVectorData(NUM_JOINTS,
                                           &cmdVecDataID,
                                           &cmdVecData);
```

```
        if (result != oSUCCESS)
        {
         OSYSLOG1((osyslogERROR, "%s : %s %d",
                  "MovingHead7::NewCommandVectorData()",
                  "OPENR::NewCommandVectorData() FAILED",
                  result));
        }
        region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
                                 cmdVecData->vectorInfo.offset,
                                 (void*)cmdVecData,
                                 cmdVecData->vectorInfo.totalSize);
        cmdVecData->SetNumData(NUM_JOINTS);
        for (int j = 0; j < NUM_JOINTS; j++)
        {
           info = cmdVecData->GetInfo(j);
           info->Set(odataJOINT_COMMAND2,
                     jointID[j],
                     ocommandMAX_FRAMES);
        }
     }
  }
```

Set the effector value

Once the memory has been allocated by creating the *RCRegion*, then joint values can be sent. The process begins by checking that no other object is reading the shared memory before writing to it. To do that, the *RCRegion* class has a function called *NumberOfReference()* that returns the number of objects pointing to that memory (this control is established when calling the *OPENR::NewCommandVectorData()* explained above). The function will return the number of objects pointing to that memory including the present object (so a value of 1 is correct to start writing).

Once it is sure to write to the region, a sequence of frame commands must be created. In order to create a linear movement from the current joint position to the desired new one, a set of mid-steps must be created. This process brings smoothness in the robot movement and prevents damages for too high velocity movements. The way of creating this mid-steps must be decided by the programmer and implemented by it. There is no help function provided for it. This allows the programmer to design his own type of movements (linear, exponential,etc.). All the frames generated will then fill the frames in the *OCommandData*.

- Example:

```
MovingResult MovingHead7::MoveToZeroPos()
{
    static int counter = -1;
    static double s_tilt1, d_tilt1;
```

```
    static double s_pan,  d_pan;
    static double s_tilt2, d_tilt2;
    if (counter == -1) {
        OJointValue current;
        OPENR::GetJointValue(jointID[TILT1_INDEX], &current);
        s_tilt1 = degrees(current.value/1000000.0);
        d_tilt1 = (TILT1_ZERO_POS - s_tilt1) /
                  (double)ZERO_POS_MAX_COUNTER;
        OPENR::GetJointValue(jointID[PAN_INDEX], &current);
        s_pan = degrees(current.value/1000000.0);
        d_pan = (PAN_ZERO_POS - s_pan) /
                (double)ZERO_POS_MAX_COUNTER;
        OPENR::GetJointValue(jointID[TILT2_INDEX], &current);
        s_tilt2 = degrees(current.value/1000000.0);
        d_tilt2 = (TILT2_ZERO_POS - s_tilt2) /
                  (double)ZERO_POS_MAX_COUNTER;
        counter = 0;
        RCRegion* rgn = FindFreeRegion();
        OSYSDEBUG(("FindFreeRegion() %x \n", rgn));
        SetJointValue(rgn, TILT1_INDEX, s_tilt1, s_tilt1 + d_tilt1);
        SetJointValue(rgn, PAN_INDEX,   s_pan,   s_pan   + d_pan);
        SetJointValue(rgn, TILT2_INDEX, s_tilt2, s_tilt2 + d_tilt2);
        subject[sbjMove]->SetData(rgn);
        s_tilt1 += d_tilt1;
        s_pan   += d_pan;
        s_tilt2 += d_tilt2;
        counter++;
    }
    RCRegion* rgn = FindFreeRegion();
    OSYSDEBUG(("FindFreeRegion()%x \n", rgn));
    SetJointValue(rgn, TILT1_INDEX, s_tilt1, s_tilt1 + d_tilt1);
    SetJointValue(rgn, PAN_INDEX,   s_pan,   s_pan   + d_pan);
    SetJointValue(rgn, TILT2_INDEX, s_tilt2, s_tilt2 + d_tilt2);
    subject[sbjMove]->SetData(rgn);
    subject[sbjMove]->NotifyObservers();
    s_tilt1 += d_tilt1;
    s_pan   += d_pan;
    s_tilt2 += d_tilt2;
    counter++;
    return (counter == ZERO_POS_MAX_COUNTER) ?
            MOVING_FINISH : MOVING_CONT;
}
...
RCRegion* MovingHead7::FindFreeRegion() {
    for (int i = 0; i < NUM_COMMAND_VECTOR; i++) {
        if (region[i]->NumberOfReference() == 1) return region[i];
```

```
    }
    return 0;
}
void MovingHead7::SetJointValue(RCRegion* rgn, int idx,
                                double start, double end)
{
    OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();
    OCommandInfo* info = cmdVecData->GetInfo(idx);
    info->Set(odataJOINT_COMMAND2, jointID[idx], ocommandMAX_FRAMES);
    OCommandData* data = cmdVecData->GetData(idx);
    OJointCommandValue2* jval = (OJointCommandValue2*)data->value;
    double delta = end - start;
    for (int i = 0; i < ocommandMAX_FRAMES; i++) {
        double dval = start + (delta * i) / (double)ocommandMAX_FRAMES;
        jval[i].value = oradians(dval);
    }
}
```

5 A simple OPEN-R controller using neural nets

In this section we will use what we have learned on previous chapters to construct a controller to move one of Aibo's leg using a simple neural network as the controller element.

5.1 Description of the problem

We want to solve the following control problem: we want to move the right-fore leg of Aibo. We want to move its three joints, making each of them to perform an oscilation, i.e., they must go to the top of its positive position and return to the minimum position, and keep on going.

To obtain this behavior we could have written a program that says at any time step the required position of the robot in order to perform an oscilation (for example calculating the required position as the sinus of the current step). Instead of that we have decided to implement a controller formed by neural networks, just to test what has been learned during the other parts of the course (see parts I and II of the course) and to compare the results.

Since there are three different joints in one leg, we will create three different neural networks, each one in charge of every joint.

The implementation problem will consist on creating an OPEN-R program that reads the sensor values of the joints, applies those values to the neural nets, and uses the nets outputs to decide the velocity that has to be applied to the joint. We will base our program on the already existing sample program MovingLegs7, plus the use of some specially prepared object that implements the neural networks.

5.2 The neural controller

The neural networks are formed of a feed forward net, with two inputs, corresponding to the current state of the joint and the state on the previous time step; five hidden units with hiperbolic tangent as activation function; one output with lineal function as activation function, that indicates the desired velocity of the joint. The output of each net indicates the velocity that is required for the joint. The time step is 128 ms.

For the construction of such controller we have created a C++ object called *FeedForwardNetwork* that implements the neural networks, including their construction, initialisation and activation functions. The *FeedForwardNetwork::LoadDescriptionFromFile* procedure takes a file containing the description of the network and populates the network attributes. The file must contain the number of input, hidden and output units, the bias that will be applied to both layers (hidden and output), and the a list of the weights of the connections of each hidden with the inputs, and the list of weightsn of the connections of the outputs with the hiddens. Those weights should have been calculated previously by other means. In our example, weights were calculated by using a group of examples under Matlab (see the Part I of this text), and stored in three different files: *aibored1.txt, aibored2.txt and aibored3.txt*.

For each a neural network will be constructed. Neural nets have 2 inputs, corresponding to the sensed value of the joint in this time-step and the value sensed in the

previous time-step. Hidden layer is composed of 5 units, and the output layer is a single neuron that provides the velocity required for the joint.

5.3 Constructing the program

The structure of the program will be the following: the Right-Fore leg of Aibo will be initialised in a special position. Then the neural controller will take control. By using the *OPENR::GetJointValue* primitive it will obtain the values of the three joints. With those values and the ones taken in the previous time-step, the neural networks associated to the joints will be activated, generating the required velocity of the joint. Once this velocity is obtained, it will be utilised to calculate the commands that need to be sent to the joints using the *OCommanVectorData* structure. Only one object will be created, named *NeuralLegControl*, that will communicate with *OVirtualRobotComm* in order to send to it new joint commands.

Configuration files

The *NeuralLegControl* object only communicates with OVirtualRobotComm, so it only has one gate, that we will call *Move*. So the *stub.cfg* file will look like this:

```
ObjectName : NeuralLegControl
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "NeuralLegControl.Move.
          OCommandVectorData.S", null,Ready()
Service : "NeuralLegControl.DummyObserver.
          DoNotConnect.O", null, null
```

And the *connect.cfg* file:

```
NeuralLegControl.Move.OCommandVectorData.S
OVirtualRobotComm.Effector.OCommandVectorData.O
```

For this program to work, only the *PowerMonitor* extra object will be required, so it will have to be compiled by the makefile, and included in the *object.cfg* file for execution.

Initialisation/Finalization functions

The *NeuralLegControl* object has 6 different states MLS_IDLE, MLS_START, MLS_ADJUSTING_DIFF_JOINT, MLS_MOVING_TO_BROADBASE, MLS_MOVING_TO_SLEEPING, MLS_NEURO_CONTROLLED.

Only in MLS_NEURO_CONTROLLED state the neurons take control of the leg. The other states are initialisation states.

The constructor of *NeuralLegControl*, creates the three neural nets, and populates them with the values obtained from files stored in */MS/OPEN-R/MW/DATA/P/* Aibo directory.

```
NeuralLegControl::NeuralLegControl() : movingLegsState(MLS_IDLE)
{
   net1 = new FeedForwardNetwork ();
   net1->LoadDescriptionFromFile
            ("/MS/OPEN-R/MW/DATA/P/aibored1.txt");
   net2 = new FeedForwardNetwork ();
   net2->LoadDescriptionFromFile
            ("/MS/OPEN-R/MW/DATA/P/aibored2.txt");
   net3 = new FeedForwardNetwork ();
   net3->LoadDescriptionFromFile
            ("/MS/OPEN-R/MW/DATA/P/aibored3.txt");
}
```

The *DoInit, DoStart, DoStop* and *DoDestroy* functions are as usual. During initialisation the primitive IDs of the joints are calculated (*OpenPrimitives()*), and the memory regions are reserved for command sending (*NewVectorData()*). Also, motors are switched on.

```
OStatus NeuralLegControl::DoInit(const OSystemEvent& event)
{
    OSYSDEBUG(("NeuralLegControl::DoInit()\n"));
    NEW_ALL_SUBJECT_AND_OBSERVER;
    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;
    OpenPrimitives();
    NewCommandVectorData();
    OPENR::SetMotorPower(opowerON) ;
    return oSUCCESS;
}
OStatus NeuralLegControl::DoStart(const OSystemEvent& event)
{
    OSYSDEBUG(("NeuralLegControl::DoStart()\n"));
    if (subject[sbjMove]->IsReady() == true)
    {
        AdjustDiffJointValue();
        movingLegsState = MLS_ADJUSTING_DIFF_JOINT_VALUE;
    }
    else
    {
        movingLegsState = MLS_START;
    }
    ENABLE_ALL_SUBJECT;
    ASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
```

```
}
OStatus NeuralLegControl::DoStop(const OSystemEvent& event)
{
    OSYSDEBUG(("NeuralLegControl::DoStop()\n"));
    movingLegsState = MLS_IDLE;
    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}
OStatus NeuralLegControl::DoDestroy(const OSystemEvent& event)
{
    DELETE_ALL_SUBJECT_AND_OBSERVER;
    return oSUCCESS;
}
```

Calculating primitives ID

```
void NeuralLegControl::OpenPrimitives()
{
    for (int i = 0; i < NUM_JOINTS; i++)
    {
        OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i],
                         &jointID[i]);
        if (result != oSUCCESS)
        {
            OSYSLOG1((osyslogERROR, "%s : %s %d",
                      "NeuralLegControl::DoInit()",
                      "OPENR::OpenPrimitive() FAILED", result));
        }
    }
}
```

Creating the memory regions

The following function will create the RCRegion memory regions that will be used to send the commands to the joints. This code creates NUM_COMMAND_VECTOR memory regions, instead of creating only one. The reason is to provide more that one region that will allow the sending of several commands even when the previous ones have not been yet processed. For this reason, a function that looks for one free memory region is also required.

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

```
    OCommandInfo*       info;
    for (int i = 0; i < NUM_COMMAND_VECTOR; i++)
    {
        OCommandVectorData result =
              OPENR::NewCommandVectorData(NUM_JOINTS,
                                          &cmdVecDataID,
                                           &cmdVecData);
        if (result != oSUCCESS)
        {
            OSYSLOG1((osyslogERROR, "%s : %s %d",
                      "NeuralLegControl::NewCommandVectorData()",
                      "OPENR::NewCommandVectorData() FAILED",
                      result));
        }
        region[i] = new RCRegion(cmdVecData->vectorInfo.memRegionID,
                                 cmdVecData->vectorInfo.offset,
                                 (void*)cmdVecData,
                                 cmdVecData->vectorInfo.totalSize);
        cmdVecData->SetNumData(NUM_JOINTS);
        for (int j = 0; j < NUM_JOINTS; j++)
        {
            info = cmdVecData->GetInfo(j);
            info->Set(odataJOINT_COMMAND2,
                      ocommandMAX_FRAMES);
        }
    }
}
RCRegion* NeuralLegControl::FindFreeRegion()
{
    for (int i = 0; i < NUM_COMMAND_VECTOR; i++)
    {
        if (region[i]->NumberOfReference() == 1)
             return region[i];
    }
    OSYSPRINT (("No free RCRegion available\n"));
    return 0;
}
```

Setting the joint gains and adjusting the difference values

The following routines set the joints gains, and adjust the diferences between what the joint sensors say and the actual position of the joint, as explained in section 4.

```
void NeuralLegControl::SetJointGain()
{
    for (int i = 0; i < 4; i++)
```

```
    {
        int j1 = 3 * i;
        int j2 = 3 * i + 1;
        int j3 = 3 * i + 2;
        OPENR::EnableJointGain(jointID[j1]);
        OPENR::SetJointGain(jointID[j1],
                            J1_PGAIN, J1_IGAIN, J1_DGAIN,
                            PSHIFT, ISHIFT, DSHIFT);
        OPENR::EnableJointGain(jointID[j2]);
                            J2_PGAIN, J2_IGAIN, J2_DGAIN,
                            PSHIFT, ISHIFT, DSHIFT);
        OPENR::EnableJointGain(jointID[j3]);
        OPENR::SetJointGain(jointID[j3],
                            J3_PGAIN, J3_IGAIN, J3_DGAIN,
                            PSHIFT, ISHIFT, DSHIFT);
    }
}
NeuralLegControl::AdjustDiffJointValue()
{
    OJointValue current[NUM_JOINTS];
    for (int i = 0; i < NUM_JOINTS; i++)
    {
        OJointValue current;
        OPENR::GetJointValue(jointID[i], &current);
        SetJointValue(region[0], i,
                      degrees(current.value/1000000.0),
                      degrees(current.value/1000000.0));
    }
    subject[sbjMove]->SetData(region[0]);
    subject[sbjMove]->NotifyObservers();
    return MOVING_FINISH;
}
```

The Ready function

The Ready function will be activated every time that an AR message is received from *OVirtualRobotComm* through the Move gate. This indicates that the last command sent to *OVirtualRobotComm* indicating to move a joint has been processed, and the virtual object is ready to receive a new command (if required). So, we will use this function to orchestrate the functioning of the object. We will first move the dog to a broadbase position and the move it to the sleeping position. Once in this position (the object will be in MLS_NEURO_CONTROLLED state) the neural controller will take control of the dog and will activate the neural nets, each time an AR is received. The activation of the neural nets generates a new command to be sent to the joints, and when this new command has been performed by the robot a new AR message will come through the *Move* gate, activating again the nets and reproducing all the time the same control

mechanism.

```
void NeuralLegControl::Ready(const OReadyEvent& event)
{
    OSYSDEBUG(("NeuralLegControl::Ready()\n"));
    if (movingLegsState == MLS_NEURO_CONTROLLED)
    {
        OSYSDEBUG(("MLS_NEURO_CONTROLLED\n"));
        GenerateNeuralControl ();
        ; // do nothing
    } else if (movingLegsState == MLS_START)
    {
        OSYSDEBUG(("MLS_START\n"));
        AdjustDiffJointValue();
        movingLegsState = MLS_ADJUSTING_DIFF_JOINT_VALUE;
    } else if (movingLegsState == MLS_ADJUSTING_DIFF_JOINT_VALUE)
    {
        OSYSDEBUG(("MLS_ADJUSTING_DIFF_JOINT_VALUE\n"));
        SetJointGain();
        MovingResult r = MoveToBroadBase();
        movingLegsState = MLS_MOVING_TO_BROADBASE;
    } else if (movingLegsState == MLS_MOVING_TO_BROADBASE)
    {
        OSYSDEBUG(("MLS_MOVING_TO_BROADBASE\n"));
        MovingResult r = MoveToBroadBase();
        if (r == MOVING_FINISH)
        {
            movingLegsState = MLS_MOVING_TO_SLEEPING;
        }
    } else if (movingLegsState == MLS_MOVING_TO_SLEEPING)
    {
        OSYSDEBUG(("MLS_MOVING_TO_SLEEPING\n"));
        MovingResult r = MoveToSleeping();
        if (r == MOVING_FINISH)
        {
            movingLegsState = MLS_NEURO_CONTROLLED;
        }
    }
}
```

MovingToBroadBase and MoveToSleeping

This functions implement the required sequence of movements to move Aibo's joints to the Broadbase and Sleeping positions. Position values are described in the *NeuralLegControl.h* file. Those functions make use of the *SetJointAngle* function to actually

send the command to the joints

```
MovingResult NeuralLegControl::MoveToBroadBase()
{
    static int counter = -1;
    static double start[NUM_JOINTS];
    static double delta[NUM_JOINTS];
    double ndiv = (double)BROADBASE_MAX_COUNTER;
    if (counter == -1) {
        for (int i = 0; i < NUM_JOINTS; i++) {
            OJointValue current;
            OPENR::GetJointValue(jointID[i], &current);
            start[i] = degrees(current.value/1000000.0);
            delta[i] = (BROADBASE_ANGLE[i] - start[i]) / ndiv;
        counter = 0;
        RCRegion* rgn = FindFreeRegion();
        for (int i = 0; i < NUM_JOINTS; i++) {
            SetJointValue(rgn, i, start[i], start[i] + delta[i]);
            start[i] += delta[i];
        }
        subject[sbjMove]->SetData(rgn);
        counter ++;
    }
    RCRegion* rgn = FindFreeRegion();
    for (int i = 0; i < NUM_JOINTS; i++) {
        SetJointValue(rgn, i, start[i], start[i] + delta[i]);
        start[i] += delta[i];
    }
    subject[sbjMove]->SetData(rgn);
    subject[sbjMove]->NotifyObservers();
    counter++;
    return (counter == BROADBASE_MAX_COUNTER) ?
              MOVING_FINISH : MOVING_CONT;
}
NeuralLegControl::MoveToSleeping() {
    static int counter = -1;
    static double start[NUM_JOINTS];
    static double delta[NUM_JOINTS];
    double ndiv = (double)SLEEPING_MAX_COUNTER;
    if (counter == -1) {
        for (int i = 0; i < NUM_JOINTS; i++) {
            start[i] = BROADBASE_ANGLE[i];
            delta[i] = (SLEEPING_ANGLE[i] - start[i]) / ndiv;
        }
        counter = 0;
    }
```

```
    RCRegion* rgn = FindFreeRegion();
    for (int i = 0; i < NUM_JOINTS; i++) {
        SetJointValue(rgn, i, start[i], start[i] + delta[i]);
        start[i] += delta[i];
    }
    subject[sbjMove]->SetData(rgn);
    subject[sbjMove]->NotifyObservers();
    counter++;
    return (counter == SLEEPING_MAX_COUNTER)
              ? MOVING_FINISH : MOVING_CONT;
}
```

The neural controller

These are the two routines that activate the neural nets and obtain the required velocity to apply to the joints. The first routine gets the neural nets output. The second one translates this output to the command sequence for the joints.

```
void NeuralLegControl::GenerateNeuralControl()
{
    std::vector<double> input(2);
    std::vector<double> output(1);
    double velocity[3];

    // activates network 1 to calculate velocity
    OJointValue current;
    OPENR::GetJointValue(jointID[0], &current);
    input[0] = (double) (current.value/1000000.0);
    input[1] = previous_sensor_value[0];
    net1->activate(input,output);
    velocity[0] = output[0];
    previous_sensor_value[0]=(double) (current.value/1000000.0);
    // activates network 2 to calculate velocity
    OPENR::GetJointValue(jointID[1], &current);
    input[0] =(double) (current.value/1000000.0);
    input[1] = previous_sensor_value[1];
    net2->activate(input,output);
    velocity[1] = output[0];
    previous_sensor_value[1]=(double) (current.value/1000000.0);
    // activates network 3 to calculate velocity
    OPENR::GetJointValue(jointID[2], &current);
    input[0] =(double) (current.value/1000000.0);
    input[1] = previous_sensor_value[2];
    net1->activate(input,output);
    velocity[2] = output[0];
    previous_sensor_value[2]=(double) (current.value/1000000.0);
```

```
    SendValueToJoint(velocity);
    return;
}
void NeuralLegControl::SendValueToJoint(double* velocity)
{
    static double start[3];
    static double delta[3];
    OJointValue current;
    RCRegion* rgn = FindFreeRegion();
    for (int i = 0; i<3; i++)
    {
        if (velocity[i] > maxJointVelocity)
             velocity[i] = maxJointVelocity;
        if (velocity[i] < minJointVelocity)
             velocity[i] = minJointVelocity;
        OPENR::GetJointValue(jointID[i], &current);
        start[i] = degrees(current.value/1000000.0);
        delta[i] = degrees(0.128*velocity[i]);
        if ( (start[i] + delta[i]) > MAX_JOINT_ANGLES[i] ||
             (start[i] + delta[i]) < MIN_JOINT_ANGLES[i])
                  delta[i] = 0;
        SetJointValue(rgn,i,start[i],start[i] + delta[i]);
    }
    subject[sbjMove]->SetData(rgn);
    subject[sbjMove]->NotifyObservers();
    return ;
}
```

The function that implements the actual command to the joints: SetJointValue

This function takes the required starting and ending position of a joint and generates the sequence of commands for the joint required to smoothly perform such movement. This function takes as argument a RCRegion and fills it with the movement values.

```
void NeuralLegControl::SetJointValue(RCRegion* rgn, int idx,
                                     double start, double end)
{
    OCommandVectorData* cmdVecData =
             (OCommandVectorData*)rgn->Base();
    OCommandInfo* info = cmdVecData->GetInfo(idx);
    info->Set(odataJOINT_COMMAND2, jointID[idx],
             ocommandMAX_FRAMES);
    OCommandData* data = cmdVecData->GetData(idx);                    O

    double delta = end - start;
```

```
    for (int i = 0; i < ocommandMAX_FRAMES; i++)
    {
        double dval = start + (delta * i)
                    / (double)ocommandMAX_FRAMES;
        jval[i].value = oradians(dval);
    }
}
```

The definitions file NeuralLegControl.h

This file contains all the definitions required for the NeuralLegControl.cc to function

```
#ifndef NeuralLegControl_h_DEFINED
#define NeuralLegControl_h_DEFINED
#include <OPENR/OObject.h>
#include <OPENR/OSubject.h>
#include <OPENR/OObserver.h>
#include "def.h"
#include "FeedForwardNetwork.h"
#define maxJointVelocity 2
#define minJointVelocity 0
enum NeuralLegControlState {
    MLS_IDLE,
    MLS_START,
    MLS_ADJUSTING_DIFF_JOINT_VALUE,
    MLS_MOVING_TO_BROADBASE,
    MLS_MOVING_TO_SLEEPING,
 MLS_NEURO_CONTROLLED };
enum MovingResult {
    MOVING_CONT,
    MOVING_FINISH };
static const char* const JOINT_LOCATOR[] = {
"PRM:/r4/c1-Joint2:41",       // RFLEG J1 (Right Front Leg)
"PRM:/r4/c1/c2-Joint2:42",    // RFLEG J2
"PRM:/r4/c1/c2/c3-Joint2:43", // RFLEG J3
"PRM:/r2/c1-Joint2:21",       // LFLEG J1 (Left Front Leg)
"PRM:/r2/c1/c2-Joint2:22",    // LFLEG J2
"PRM:/r2/c1/c2/c3-Joint2:23", // LFLEG J3
"PRM:/r5/c1-Joint2:51",       // RRLEG J1 (Right Rear Leg)
"PRM:/r5/c1/c2-Joint2:52",    // RRLEG J2
"PRM:/r5/c1/c2/c3-Joint2:53", // RRLEG J3
"PRM:/r3/c1-Joint2:31",       // LRLEG J1 (Left Rear Leg)
"PRM:/r3/c1/c2-Joint2:32",    // LRLEG J2
"PRM:/r3/c1/c2/c3-Joint2:33"  // LRLEG J3 };
const double BROADBASE_ANGLE[] = {
    120,  // RFLEG J1
```

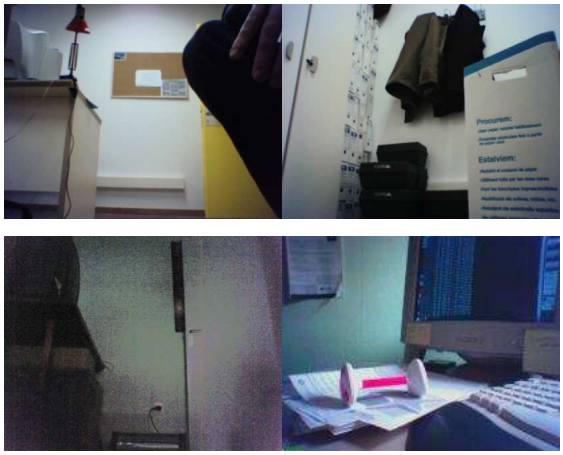
```
    90,   // RFLEG J2
    30,   // RFLEG J3
    120,  // LFLEG J1
    90,   // LFLEG J2
    30,   // LFLEG J3
   -120, // RRLEG J1
    70,   // RRLEG J2
    30,   // RRLEG J3
    -120, // LRLEG J1
    70,   // LRLEG J2
    30    // LRLEG J3 };
const double SLEEPING_ANGLE[] = {
    59,   // RFLEG J1
    0,    // RFLEG J2
    30,   // RFLEG J3
    59,   // LFLEG J1
    0,    // LFLEG J2
    30,   // LFLEG J3
    -119, // RRLEG J1
    4,    // RRLEG J2
    122,  // RRLEG J3
    -119, // LRLEG J1
    4,    // LRLEG J2
    122   // LRLEG J3 };
const double MAX_JOINT_ANGLES[] = { // in radians
   2.27,   //J1
   1.536,  //J2
   2.13    //J3 };
const double MIN_JOINT_ANGLES[] = { // in radians
   -2.01,  //J1
   -0.175, //J2
   -0.437  //J3 };
class NeuralLegControl : public OObject {
public:
    NeuralLegControl();
    virtual ~NeuralLegControl() {}
    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 Ready(const OReadyEvent& event);
    void NotifyERS7(const ONotifyEvent& event);
private:
        FeedForwardNetwork* net1;
```

```
            FeedForwardNetwork* net2;
            FeedForwardNetwork* net3;
    void          OpenPrimitives();
    void          NewCommandVectorData();
    void          SetJointGain();
    MovingResult  AdjustDiffJointValue();
    MovingResult  MoveToBroadBase();
    MovingResult  MoveToSleeping();
    void  ActivateNeuroControl(OSensorFrameVectorData* sensorVec);
    void                    GenerateNeuralControl();
    void InitERS7SensorIndex(OSensorFrameVectorData* sensorVec);
    RCRegion* FindFreeRegion();
    void SetJointValue(RCRegion* rgn, int idx,
                       double start, double end);
    void SendValueToJoint( double* velocity);
    static const size_t NUM_JOINTS         = 12;
    static const size_t NUM_COMMAND_VECTOR = 2;
    static const word   J1_PGAIN = 0x0010;
    static const word   J1_IGAIN = 0x0004;
    static const word   J1_DGAIN = 0x0001;
    static const word   J2_PGAIN = 0x000a;
    static const word   J2_IGAIN = 0x0004;
    static const word   J2_DGAIN = 0x0001;
    static const word   J3_PGAIN = 0x0010;
    static const word   J3_IGAIN = 0x0004;
    static const word   J3_DGAIN = 0x0001;
    static const word   PSHIFT   = 0x000e;
    static const word   ISHIFT   = 0x0002;
    static const word   DSHIFT   = 0x000f;
    static const int BROADBASE_MAX_COUNTER = 24;
    static const int SLEEPING_MAX_COUNTER  = 24;
    NeuralLegControlState          movingLegsState;
    OPrimitiveID               jointID[NUM_JOINTS];
    bool                           initSensorIndex;
    bool                                    prueba;
    double                previous_sensor_value[3];
    int                  ers7idx[NUM_ERS7_SENSORS];
    RCRegion*           region[NUM_COMMAND_VECTOR];
};
#endif // NeuralLegControl_h_DEFINED
```

6 Accessing the camera using OPEN-R

Eventhough Aibo's camera can be thought as a sensor, it was not introduced on section 3 for its complexity and differences with normal sensors.

Reading images from Aibo's camera can be done by obtaining information from 4 different layers. Each of these layers sends different information and is the programmer the one who should choose which layer to listen to. First three layers correspond to the actual image that the camera is obtaining, but in three different resolutions (high, medium and low). Fourth layer correspond to a segmentation layer, where dedicated hardware in Aibo performs a segmentation by colors. This text will only cover the handling of the 3 first layers.



Some images taken by the Aibo camera

Images received from the camera are in YCrCb format. This means that each pixel is composed of three values: luminance (Y), red minus luminance (Cr) and blue minus luminance (Cb). Each value is contained on a byte. The programmer should obtain those three values and then make himself the mixture in order to obtain the real YCrCb value of the pixel.

In order to obtain an image from the camera, the following steps will be required: first, camera configuration, second, to obtain a message with the image from OVirtualRobotComm. Then the layer required will be selected and it will follow the obtention of the color bands information.

6.1 Camera configuration

Light conditions affect very much the images obtained by the camera. For this reason it will be required to configure the camera properties to obtain a good capture.

Parameters that can be configured are:

- gain: the parameter to modify is called *oprmreqCAM_SET_GAIN* and it could have three posible values: *ocamparamGAIN_HIGH* for a high value, *ocamparamGAIN_MID* for a medium value, and *ocamparamGAIN_LOW* for a low value.

- white balance: the parameter to modify is called *oprmreqCAM_SET_WHITE_BALANCE* and has other three possible values: *ocamparamWB_INDOOR_MODE*, *ocamparamWB_OUTDOOR_MODE* and *ocamparamWB_FL_MODE*, for indoor, outdoor or fluorescent light conditions

- shutter: the parameter to modify is called *oprmreqCAM_SET_SHUTTER_SPEDD* and has other three values: *ocamparamSHUTTER_FAST*, *ocamparamSHUTTER_MID* and *ocamparamSHUTTER_SLOW*, for fast, medium and slow values.

On indoor aplications, best results are obtained with high gain,and low shutter speed.

6.2 Obtaining the camera message

Once the camera is configured, you can capture images comming from the *FbkImageSensor* gate of *OVirtualRobotComm*. Format of this message is *OFbkImageVectorData* and can be received by any OPEN-R object listening to that gate.

To stablish a connection between our OPEN-R object and the image gate, you must declare first an input gate in your object through where the image message will come (this has to be specified in the *stub.cfg* file of your object), and then you must specify the connection between your object and the *OVirtualRobotComm* in the *connect.cfg* file.

So for the *stub.cfg* file you must have a line like this:

```
Service: "yor_object_name.your_object_gate.OFbkImageVectorData.O",
          null, your_image_processing_routine()
```

And for the connect.cfg:

```
OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S
your_object_name.your_object_gate.OFbkImageVectorData.O
```

The message received is of type *OFbkImageVectorData*. This message is a structure containing three members:

- *vectorInfo* member of type *ODataVectorInfo*

- an array of data type *OFbkImageInfo*. This array is of size 4, corresponding each one to one of the image layers. Each member of this array is associated to one byte of the next array of bytes (see below). The associated byte is a pointer to the image data for that layer. To acces to one of those elements, you must use the *GetInfo()* function

- an array of bytes. This is an array of 4 bytes, each one correspond to one of the *OFbkImageInfo* elements of the previous array. This byte is a pointer to the image data of a layer. To access to one of those bytes you must use the *GetData()* function.

To select the element of the arrays you want to access, you should use one of the following indexes:

- o*fbkimageLAYER_H* for color high resolution image
- *ofbkimageLAYER_M* for color medium resolution image
- *ofbkimageLAYER_L* for color low resolution image
- *ofbkimageLAYER_C* for color detection image

Examples: GetInfo(ofbkimageLAYER_H) or GetData(ofbkimageLAYER_C)

6.3 Accessing the layers and the image's bands

Once we have the pointers to the layer selected, we can actually access to the image data; but remember that image data is divided into four bands. So it will be necessary to access each of those bands and merge them in order to have a complet image. To access each one of the bands we will need to use the following pointers:ofbkimageBAND_Y, ofbkimageBAND_Cr, ofbkimageBAND_Cb and ofbkimageBAND_CDT.

The OPEN-R class that handles image data is OFbkImage. To obtain one of those objects from a message received from OVirtualRobotComm containing image data, we need to create an instance of it, especifying the layer and band to access in the following way:

OFbkImage (message_data->GetInfo(ofbkimageLAYER_M),
message_data->GetData(ofbkimageLAYER_M),
ofbkimageBAND_Y)

You should retrieve the three components of every image (Y,Cr and Cb) and then merge them by hand, in order to obtain a real image. The format you choose for your real image will determine how you must combine the three bands. OPEN-R does not provide any method to do this, so the user must implement it by himself.

When accessing images throught the OFbkImage class, OPEN-R provides of some useful functions in order to facilitate the access to the image data. Some of them are:

- bool IsValid(): returns true if the OFbkImage is a valid one.
- byte* Pointer(): returns a pointer to the image data.
- int Width(): returns the image width.
- int Height(): returns the image height.
- byte Pixel(int x, int y): returns the value of the pixel (x,y).

Example: the ImageObserver sample program. Obtaining each band of an image

```
byte pixels[3];
OFbkImageInfo* info = imageVec->GetInfo(layer);
byte* data = imageVec->GetData(layer);
```

```
OFbkImage yImage(info, data, ofbkimageBAND_Y);
OFbkImage crImage(info, data, ofbkimageBAND_Cr);
OFbkImage cbImage(info, data, ofbkimageBAND_Cb);
```

7 Webots integration with Aibo

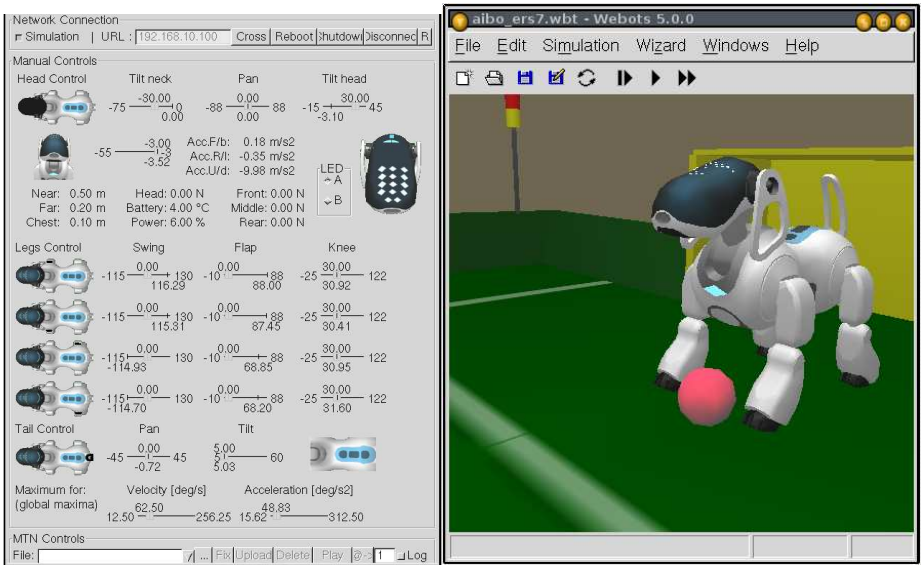
This section explains how to use Aibo together with the Webots simulator software.

Webots is a simulator software for any type of robot, including custom ones. Nevertheless, it contains some specially prepared packages to work together with the Aibo robot. The first package, allows Webots the remote control and monitoring of the Aibo robot by connecting wirelessly the computer running Webots with the Aibo robot running a special server. The second package allows the cross-compilation of a Webots controller into Aibo code. This means that you can develop and test your controller on the simulator, and once you are happy with the results, you can cross-compile that controller into an OPEN-R controller that will be executed on the real robot autonomously.

Webots can handle up to date two different models of Aibo: the ERS-210 and the ERS-7.

7.1 Remote control and monitoring

Webots allows the remote control and monitoring of the Aibo robot by means of a control pannel (called the client) and a OPEN-R program on Aibo (called the server). The OPEN-R program is installed on a memory stick an executed on Aibo. It accepts connections from the control panel that is running on the computer equiped with a wireless card. The control panel sends commands to the server to obtain the robot status and to give orders to the robot. The resulting state is presented on the control panel.



The control panel allows for a control of both simulator and real robot at the same time. You can move a joint, switch on-off the LEDs and plungers, or even execute

MTN files (at present partially supported on ERS7), and all those things can be done on the simulated robot, the real robot, or both.

7.1.1 Instalation of the server in Aibo

To install the server in Aibo, just copy the webots/xxxx/OPEN-R directory into a previously prepared memory stick (see section 1 for stick preparation). Configure the LAN settings accordingly with your wireless LAN, and insert it into you robot.

7.1.2 Network functions

The first part of the control pannel handles network functions. If the pannel is not connected to the real robot, all modifications to the pannel settings will affect only to the simulated robot.

The network buttons allows for the connection, disconnection and reboot to/of the robot. There is also a special button that hooks simulation with the real robot. When this button is on, all modifications to the controls will affect both simulator and real robot. If switched off, mods will only affect the real robot.

7.1.3 Manual controls and feedback

The central part of the pannel is dedicated to the presentation of sensor values. There can be seen all sensor values of the simulator, when not connected to the real robot, or of the real robot, when connected. This includes distance sensors, joint sensors, preassure sensors, accelerators, as well as other internal sensors. It also contains some sliders that allow the modification of the different joinst of the robot.

This part of the pannel also permits the switching of the LEDs (face, head and back).

At the bottom of this part there are two sliders that allow the change of the maximum velocity and acceleration of the joints. This is an important parameter since it determines the movement of the robot.

7.1.4 Motion sequence playback (MTN)

MTN is Sony's format for a frame-by-frame motion sequence playback. MTN files contain data describing the position of each joint and LED of the robot at any given instant for a desired movement. Webots comes with several MTN files performing different movements, like for example dancing, walking, etc. But the user can desing his own MTN files by using the MEdit tools provided by Sony free of charge.

The MTN controls of the pannel allow for the selection, upload and playback of MTN files in both the simulation and the real robot.

7.2 Cross-compilation

This section explains how to use the cross-compilation feature in Webots. It is important to note that Webots uses the OPEN-R compiler provided by Sony in order to generate the cross-compiled controller. For this reason, it is necessary to have the

OPEN-R environment installed in your system to access this cross-compilation feature in Webots. Remember that once the controller has been cross-compiled and installed in Aibo, it does not require Webots running on a computer. The robot will act autonomously following the controller cross-compiled.

To explain the cross-compilation, we will use a simple controller developed in Webots and transfer it to the real robot. The controller program selected is *ers7_mimic*. This controller releases the right front leg of the robot for easy movement with hands of the programmer. Every movement applied to that released leg will be mimic by the other legs. The code of the controller is posted bellow.

```
#include <device/robot.h>
#include <device/servo.h>
/* this controller is quite entertaining: releasing the servos on one of
#define SIMULATION_STEP 16
#define NUM_JOINTS_PER_LEG 3
enum { /* leg indices enumeration */
LFLEG, /* left fore leg */
LHLEG, /* left hind leg */
RFLEG, /* right fore leg */
RHLEG, /* right hind leg */
NUM_LEGS};
#define MASTER_LEG RFLEG /* master leg */
static DeviceTag leg_servos[NUM_LEGS][NUM_JOINTS_PER_LEG];
static void init(void)
{

int j; leg_servos[LFLEG][0] =   robot_get_device("PRM:/r2/c1-Joint2:21");

leg_servos[LFLEG][1] = robot_get_device("PRM:/r2/c1/c2-Joint2:22");

leg_servos[LFLEG][2] = robot_get_device("PRM:/r2/c1/c2/c3-Joint2:23");

leg_servos[LHLEG][0] = robot_get_device("PRM:/r3/c1-Joint2:31");

leg_servos[LHLEG][1] = robot_get_device("PRM:/r3/c1/c2-Joint2:32");

leg_servos[LHLEG][2] = robot_get_device("PRM:/r3/c1/c2/c3-Joint2:33");

leg_servos[RFLEG][0] = robot_get_device("PRM:/r4/c1-Joint2:41");

leg_servos[RFLEG][1] = robot_get_device("PRM:/r4/c1/c2-Joint2:42");

leg_servos[RFLEG][2] = robot_get_device("PRM:/r4/c1/c2/c3-Joint2:43");

leg_servos[RHLEG][0] = robot_get_device("PRM:/r5/c1-Joint2:51");
```

```
leg_servos[RHLEG][1] = robot_get_device("PRM:/r5/c1/c2-Joint2:52");

leg_servos[RHLEG][2] = robot_get_device("PRM:/r5/c1/c2/c3-Joint2:53");
/* release master leg servos, enable position reading */
  for(j=0; j<NUM_JOINTS_PER_LEG; j++)
  {
    servo_motor_off(leg_servos[MASTER_LEG][j]);

servo_enable_position(leg_servos[MASTER_LEG][j],SIMULATION_STEP);
  }
}
static void die(void)
{ }
static int run(int ms)
{
  int i,j;
  float master;
  for(j=0; j<NUM_JOINTS_PER_LEG; j++)
  {

master = servo_get_position(leg_servos[MASTER_LEG][j]); /* master positio
    for(i=0; i<NUM_LEGS; i++)
      if( i!=MASTER_LEG )

servo_set_position(leg_servos[i][j],master); /* set remaining legs to mas
  }
  return SIMULATION_STEP;
}
int main()
{
  robot_live(init);
  robot_die(die);
  robot_run(run);
  return 0;
}
```

This controller can be normally compiled to obtain a Webots controller by doing a make. However, if cross-compilation is required, it will be necessary to do the compilation of another way.

Assuming that your controller *ers7_mimic.c* is stored in the *webots/controllers/ers7_mimic* directory,you will find three other files on that directory:

Makefile, is the makefile used for the compilation of the Webots controller.

Makefile.openr is the makefile that will be used for the cross-compilation of the Aibo controller

Makefile.sources, contains a list of the source files required for the cross-compilation of the controller for Aibo. In this case it only should contain one file (*ers7_mimic.c*)

You can copy and rehuse without change the *Makefile.openr* source on other controllers to be cross-compiled, but you will have to change the content of the *Makefile.sources*.

In order to cross-compile the controller, just type:

```
> make -f Makefile.openr
```

This will cross-compile the controller and create an OPEN-R directory in the controller directory. This OPEN-R directory contains the controller for the Aibo robot. You must merge this directory with an already prepared memory stick (see section 1) and then insert it into Aibo to run the cross-compiled controller.

You can clean the OPEN-R directory created in the Webots controller directory by typping:

```
> make -f Makefile.openr clean
```

8 Bibliography

This chapter contains a list of interesting reading related to the programming of Aibo using OPEN-R.

1. Section “Frequently Asked Questions (FAQ)” on openr.aibo.com
2. “OPEN-R SDK Documents English” on openr.aibo.com
3. “Installation guide” on openr.aibo.com
4. “Programer’s guide” on openr.aibo.com
5. “Model Information for ERS-7” on openr.aibo.com
6. “Level2Reference Guide” on openr.aibo.com
7. “OPEN-R Internet Protocol Version4” on openr.aibo.com
8. “OPEN-R SDK school” on openr.aibo.com
9. F. Martín-Rico, R. González-Careaga, J.M. Cañas, and V. Matellán, “Programming model based on concurrent objects for the AIBO robot”, XII Jornadas de Concurrencia y Sistemas Distribuídos, 2004
10. Michel, O., “Webots: Professional mobile robot simulation” , International Journal of Advanced Robotics Systems, 1-1, 2004