Informe Técnico ICC 2008-05-02

Adding camera functions to the Webots OPEN-R wrapper object for Aibo robots

Juan C. Peris, Jorge Grande and M. T. Escrig
Cognition for Robotics Research

Mayo 2008

Departamento de Ingeniería y Ciencia de los Computadores

Correo electrónico: jperis@lsi.uji.es, jorge.grande@uji.es, escrigm@icc.uji.es

Universidad Jaime I
Campus de Riu Sec, s/n
12.071 - Castellón
España

Adding camera functions to the Webots OPEN-R wrapper object for Aibo robots

Juan C. Peris[1], Jorge Grande[2] and M. T. Escrig[3]
Cognition for Robotics Research
Jaume I University

Abstract

We use the Webots™ mobile robot simulation software for testing controllers and for using a common interface for all our robotic platforms. In the case of the Aibo™ robots we use cross-compilation to combine the Webots™ controller programs with an OPEN-R wrapper object in order to obtain binary code executable on the live robot. The problem we have found is that the OPEN-R wrapper object delivered with Webots™ does not have implemented the functions for accessing the Aibo™ camera. The objective of the present work is the implementation of these camera functions in the wrapper object.

[1] Departamento de Lenguajes y Sistemas Informáticos
E-mail: jperis@lsi.uji.es
[2] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: jorge.grande@uji.es
[3] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: escrigm@icc.uji.es

Implementación de funciones para la cámara del Aibo en el compilador cruzado OPEN-R/Webots

Juan C. Peris[4], Jorge Grande[5] y M. T. Escrig[6]
Cognition for Robotics Research
Universidad Jaume I

Resumen

En nuestro grupo de investigación C4R2 utilizamos el simulador de robots móviles Webots™ para probar los controladores y poder utilizar un interfaz común para todas nuestras plataformas robóticas. En el caso de los robots Aibo™ utilizamos una compilación cruzada para combinar los controladores programados en Webots™ con los objetos OPEN-R para obtener el código binario ejecutable en los robots Aibo™ reales. El problema con el que nos hemos encontrado es que el compilador cruzado para OPEN-R de Webots™ no tiene implementadas las funciones para acceder a la cámara del Aibo™. El objetivo del presente trabajo es la implementación de estas funciones de la cámara en el compilador cruzado.

[4] Departamento de Lenguajes y Sistemas Informáticos
E-mail: jperis@lsi.uji.es
[5] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: jorge.grande@uji.es
[6] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: escrigm@icc.uji.es

Contents

LIST OF FIGURES **7**

ABOUT TRADEMARKS **9**

CHAPTER 1. INTRODUCTION **11**

1.1 AIBO 11
OPEN-R: *11*
1.2 WEBOTS 12
Cross-compilation of Webots controllers for Aibo robots *13*
1.3 PROJECT OBJECTIVES 13

CHAPTER 2. SYSTEM OVERVIEW **15**

2.1 OPEN-R 15
2.1.1 Features of OPEN-R: *15*
The virtual objects OVirtualRobotComm and OvirtualAudioComm *15*
2.1.2 Inter-object communication *16*
2.1.3 OPEN-R objects description *17*
2.1.4 Programming in OPEN-R *18*
2.1.5 The stub.cfg config file *19*
2.1.6 The connect.cfg config file *21*
2.1.7 The Notify() function *22*
2.2 AIBO'S CAMERA 22
2.2.1 Accessing the camera *22*
2.2.2 Format type of the camera data *22*
2.3 WEBOTS CROSS-COMPILATION FOR AIBO ROBOTS 23

CHAPTER 3. ADDING CAMERA FUNCTIONS TO THE WEBOTS OPEN-R WRAPPER OBJECT FOR AIBO ROBOTS **27**

3.1 CONTROLLER.H 27
3.2 CONTROLLER.CC 27
3.2.1 Notify method *27*
3.2.2 camera_enable and camera_disable *28*
3.2.3 camera_get_width and camera_get_height *28*
3.2.4 camera_get_image *29*
3.2.5 camera_get_fov *31*
3.3 STUB.CFG 31
3.4 CONNECT.CFG 32

REFERENCES **33**

ANNEX I. EXAMPLE OF A WEBOTS CONTROLLER FOR AIBO ROBOTS **35**

List of Figures

FIGURE 1. AIBO ERS-7 ROBOT. 11
FIGURE 2: WEBOTS WITH THE AIBO_ERS7.WBT WORLD 12
FIGURE 3: INTER-OBJECT COMMUNICATION 16
FIGURE 4: OPEN-R OBJECT EXAMPLE. 18
FIGURE 5: EXAMPLE OF AN INTER-OBJECT COMMUNICATION. 21
FIGURE 6. OFBKIMAGEVECTORDATA STRUCTURE 23
FIGURE 7. AIBO CROSS-COMPILATION. 24
FIGURE 8. OPEN-R DIRECTORY. 24
FIGURE 9. THE WEBOTS/TRANSFER/OPENR DIRECTORY. 25

About trademarks

- Aibo™ is a registered trademark of SONY Corporation.
- Memory Stick™ is a trademark of SONY Corporation.
- Webots™ is a registered trademark of Cyberbotics Ltd.
- Matlab™ is a registered trademark of The MathWorks, Inc.
- Mac OS X™ is registered trademark of Apple Computer, Inc. in the United States and/or other countries.
- UNIX™ is a registered trademark of The Open Group in the United States and/or other countries.
- Linux™ is a registered trademark of Linus Torvalds.
- Windows™ is registered trademark of Microsoft Corporation in the United States and/or other countries.
- MIPS™ is a registered trademark of MIPS Technologies, Inc. in the United States and/or other countries.
- Other system names, product names, service names and firm names contained in this document are generally trademarks or registered trademarks of respective makers.

Chapter 1

Introduction

1.1 Aibo

Aibo is a four-legged dog-like entertainment robot developed by Sony (*www.aibo.com*). While it is primarily intended for use as a toy, its flexibility in design and the ability to program on-board software using a C++ API (called OPEN-R) makes the Aibo a particularly interesting object for robotic research as well. In particular, the Robocup Sony Four-Legged Robot League (*www.openr.org/robocup*) is very popular among roboticists throughout the world.

These robots have a MIPS R4000 processor and a number of sensors and actuators that can be manipulated by software. Each leg has three motors (three degrees of freedom). The head has also three degrees of freedom and the tail and the ears have two degrees of freedom each one. Therefore Aibo robots can make a great variety of movements.

Aibo robots have LEDs in their head that can be used to express emotions. Moreover it has two infrared sensors, vibration and temperature sensors and a color camera.

The ERS 210 Aibo model has a slot which allows the use of a wireless card supporting the 802.11b protocol of the IEEE. In the more recent Aibo ERS 7 model this feature is built-in. This card allows a wireless communication with the robot at 11Mbps. It is possible to use the TCP/IP and UDP protocols, or a telnet connection to the Aibo port 59000.

The Aibo robot has also a slot for inserting a Memory Stick card (data storage device). Inside this Memory Stick users can program the behaviours of the robot.

Figure 1. Aibo ERS-7 robot.

OPEN-R:

Sony provides the OPEN-R SDK for developing software for the Aibo robots using the C++ language. The user can program the movements of the robot, as well as to obtain data from the sensors. Even that this development kit is mainly targeted to the Linux platform, it is also possible to use it on Windows platforms by installing the Cygwin application.

OPEN-R is a run time system built using the GCC C++ compiler targeted for the MIPS R4000 processor. The central idea of OPEN-R is that a program is built using one or more OPEN-R objects, which incorporate a state machine and send messages to each other.

The usual procedure to work with Aibo robots is the following: a program is created on a PC computer using OPEN-R and C++. This program is then compiled in the PC to target the Aibo platform. Then, the resulting binaries are transferred from the PC to a memory stick, which is inserted into the Aibo. Finally, when the robot is swiched on, the code is loaded and executed.

1.2 Webots

Webots (*http://www.cyberbotics.com*), is a three dimensional mobile robot simulation software created by Cyberbotics Ltd. It was initially developed at the Laboratoire de Micro-Informatique (LAMI) of the Swiss Federal Institute of Technology, Lausanne, Switzerland (EPFL). It is available for Linux i386, Mac OS X and Windows. It allows the user to:

- Model and simulate any type of mobile robot (wheeled, legged, winged) in a complete world with possibly light, obstacles and water using OpenGL and the Open Dynamics Engine library (ODE) for realistic physics simulation.
- Program the robots in C, C++ and Java or from third party software (like for example Matlab) through TCP/IP.
- Transfer a shipped or a self-programmed controller to a real mobile robot.

The Webots software is shipped with a simulation of the Aibo robot models ERS-210 and ERS-7 (worlds aibo_ers210.wbt and aibo_ers7.wbt).

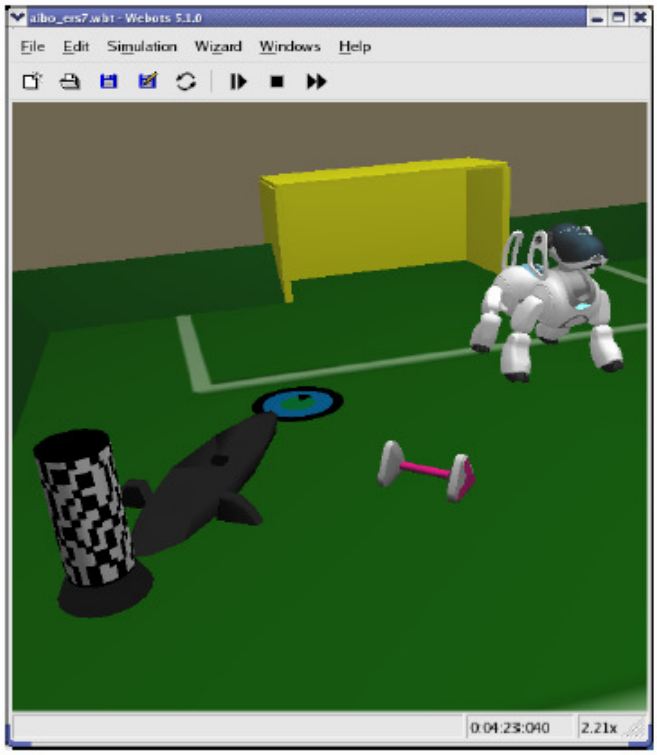

Figure 2: Webots with the aibo_ers7.wbt world

Furthermore, the simulation model includes a tool that allows one to remotely control the Aibo, both the simulated model and the real Aibo either separately or simultaneously. This tool is simply called remote control. When using it with a real Aibo, the communication is achieved through its wireless LAN interface. When using it with the simulated version, the commands are directly sent through software. The remote control is accessible by double-clicking on the Aibo model in the main Webots window showing the 3D world.

In order to achieve the communication between Webots and the real Aibo, special software must be running on Aibo. This software was programmed in OPEN-R and is called RCServer. Once Webots is running on the client computer, Aibo is up and running the RCServer and both "see" each other on the network, a connection can be established in order to remotly control the real Aibo robot.

Cross-compilation of Webots controllers for Aibo robots

Webots has also the possibility of transfer the code of robot controllers to some real robots using intermediate libraries and/or applications. For Aibo robots Webots allows the use of cross-compilation, wherein the controller code is cross-compiled to produce a binary executable which then runs on the live robot directly. You can develop and test your controllers on the simulator, and once you are happy with the results, you can cross-compile that controller into OPEN-R code that will be executed on the real robot.

Software for the Aibo robot is written using Sony's proprietary object-oriented API called OPEN-R. The basic idea of cross-compilation is this: an OPEN-R wrapper object running on Aibo basically translates all Webots robot controller API calls into OPEN-R meaningful instructions for the live robot. Because OPEN-R programs are written in C++, it is actually rather straightforward to combine the Webots controller program with the OPEN-R wrapper object to obtain a binary code executable on the live robot.

1.3 Project objectives

We are working in the Cognition for Robotics Research group (C4R2) in the Jaume I University. In our laboratory we have different robots (pioneer robots, aibo robots, a six-legged robot called Lauron IV, Khepera robots). We use the Webots software to test controllers before using the real robots and to use a common interface for all the robotic platforms.

In the case of the Aibo robots we use cross-compilation to translate all Webots controllers into OPEN-R code for the live robot. As we have mentioned in the previous point, there is an OPEN-R wrapper object which runs on Aibo. The problem we have found is that the wrapper object does not have implemented the functions of the camera. The objective of the present work is the implementation of these functions in the wrapper object.

First, in section 2 we will see an overview of the basic concepts needed to understand the work done in this technical report. After that, in section 3 the functions added to the wrapper object will be explained. Finally, in the attached document, a base code using the added camera capabilities is shown.

Chapter 2

System Overview

In this section we will explain some important concepts needed to develop the present project.

2.1 OPEN-R

"OPEN-R" is the interface promoted by Sony to expand the capabilities of the entertainment robot systems. "OPEN-R SDK" discloses the specifications of the interface between the system layer and the application layer.

2.1.1 Features of OPEN-R:

Modularised software and inter-object communication: OPEN-R software is object-oriented and modular. Software modules are called "objects" (specifically, "OPEN-R objects"). Processing is performed by multiple objects with various functionalities running concurrently and communicating via inter-object communication.

An OPEN-R object is not an object in the traditional C++ sense of the word. The concept of an object is similar to one of a process in the UNIX or Windows operating systems with regard to the following points of view:

- An object corresponds to one executable file.
- Each object runs concurrently with other objects.

Connections between objects are defined in external description files. When the system software boots, these description files are loaded and used to allocate and configure the communication paths for inter-object communication. Connection ports in objects are identified by the service name, which enables objects to be highly modular and easily replaceable as software components.

Layered structure of the software and services provided by the system layer: The OPEN-R system layer provides a set of services (input of sound data, output of sound data, input of image data, output of control data to joints, and input of data from various sensors) as the interface to the application layer. This interface is also implemented by inter-object communication.

OPEN-R services enable application objects to use the robot's underlying functionalities, without requiring detailed knowledge of the robot hardware.

The system layer also provides the interface to the TCP/IP protocol stack, which enables programmers to create networking applications utilizing the wireless LAN. The IPStack is an OPEN-R system layer object. Objects can use the network services offered by the IPv4 protocol stack by communicating with the protocol stack through normal message passing, i.e. by sending special messages to and receiving special messages from the IPStack.

The virtual objects OVirtualRobotComm and OvirtualAudioComm

The OPEN-R SDK provides two special objects (virtual objects), which provide an interface to Aibo's hardware:

OVirtualRobotComm interfaces with the dog's joints, sensors, LEDs and camera.

OVirtualRobotAudioComm interfaces with the robot audio devices.

The use of those objects is the same as the use of user defined objects by means of communication gates. The gates in those objects are already predefined to send messages to and receive messages from them.

2.1.2 Inter-object communication

Each OPEN-R object can communicate with other OPEN-R objects in order to make tasks more complex. This type of communication is known as "inter-object communication". The connections between objects have to be defined in two files, called *stub.cfg* and *connect.cfg*. These files are used by the system for making the connections between the objects.

The objects are classified in two categories: ***observers*** and ***subjects***. The objects are observers when they receive messages from other objects. The objects are subjects when they send messages to other objects. The objects can simultaneously be subjects and observers.

When two objects communicate, the side that sends data is the "subject," and the side that receives data is the "observer". The subject sends a 'NotifyEvent' to the observer. 'NotifyEvent' includes the data that the subject wants to send to the observer. The observer sends a 'ReadyEvent' to the subject. The purpose of ReadyEvent is to inform the subject that the observer is ready to receive data or not. If the observer is not ready to receive data, the subject does not send any data to the observer.

Figure 3 shows a case where the subject of object A communicates with the observer of object B.

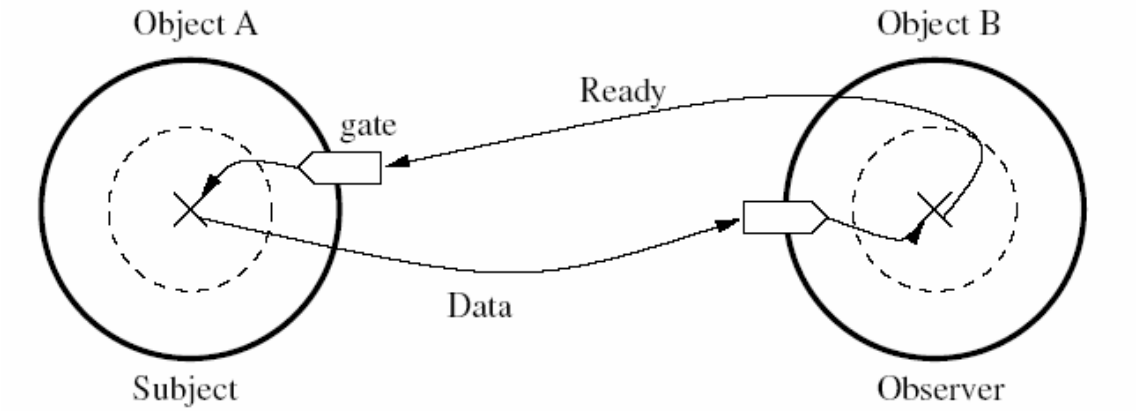

Figure 3: Inter-object communication

Before the observer receives data from the subject, the observer must inform the subject of its current state. When the observer is in a state ready to receive data, the observer sends 'ASSERT-READY' to the subject. When the observer is in a state not ready to receive data, the observer sends 'DEASSERT-READY' to the subject. When the subject receives ASSERT-READY from the observer, the subject starts to send data to the observer. After the observer receives this data and is ready to receive the next data, the subject sends ASSERT-READY again.

When the AIBO is switched on, all the objects are loaded into the system and all the connections between subjects and observers are established. The processing of each object is strictly sequential. This means that although an object can be an observer of several connections, only can process a message simultaneously. All the messages that arrive are placed in tails.

2.1.3 OPEN-R objects description

An OPEN-R object is represented with a core class (a C++ class). Each object should be represented by only one core class. The characteristics of core classes are:

- A core class inherits from the OObject class.
- A core class implements DoInit(), DoStart(), DoStop(), DoDestroy().
- A core class has the necessary number of OSubjects and OObservers.
- Some member functions in the core class correspond to specific methods in the object:
    - Methods that are called at start up and shutdown:
        - ***Init method***. This is called at start-up. This method initializes instances and variables.
        - ***Start method***. This is called at start-up after Init is executed in all objects.
        - ***Stop method***. This is called at shutdown.
        - ***Destroy method***. This is called at shutdown after Stop is executed in all objects. This method destroys the subject and observer instances.

      The Init method, Start method, Stop method, and Destroy method correspond to each DoInit(), DoStart(), DoStop() and DoDestroy() function in the object's corresponding core class, respectively.
    - When a message is received from another object, the following methods are used:
        - Methods used in subjects:
            - ***Control method***. This receives the connection results between the subject and its observers.
            - ***Ready method***. The subject receives ASSERT-READY or DEASSERT-READY notifications from the observers.
        - Methods used in observers:
            - ***Connect method***. This receives the connection results between an observer and its subjects.
            - ***Notify method***. This receives a message from the subject.

      These methods have the following characteristics:
            - The member functions corresponding to Control methods, Ready methods, Connect methods, and Notify methods are described in stub.cfg.

- The member functions receiving a message are described in stub.cfg, but it is not necessary to describe the member functions sending a message in stub.cfg.

Below, we will show an example to illustrate these concepts. The SampleClass inherits from OObject. The four standard OPEN-R member functions (DoInit(), DoStart(), DoStop() and DoDestroy()) are defined here. The object communicates with other objects and we must define the necessary number of OSubjects and OObservers. The necessary number has to be described in *def.h*.

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

class SampleClass: public OObject {
public:
        SampleClass2();
        virtual ~SampleClass2() {}

        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);

        //Describe the member functions corresponding to Notify,
        //Control, Ready, Connect method.
};
```

Figure 4: OPEN-R object example.

2.1.4 Programming in OPEN-R

- 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 flow of development of an OPEN-R program is the following (for a more detailed description see the OPEN-R SDK Programmer's guide [*OPEN-R SDK Programmer's Guide*. 2004, Sony Corporation]:
  - Design of the objects and the communication between them.
  - The connection between the entry points of an object and the actual member functions is described in an external file called ***stub.cfg***.
  - Implementation of the objects by means of core C++ classes.
  - Decide the configuration of the ***.ocf*** file. This file is used to specify the configuration of the object.
  - The objects are compiled and linked in the host machine producing executable code. There is one executable file (***.bin***) for each object.
  - Edit the setting files: OBJECT.CFG, CONNECT.CFG and DESIGNDB.CFG

- The executable code is then transferred to a memory stick that is inserted in the Aibo robot and executed on it. When Aibo boots, all the compiled objects are loaded into memory and started as concurrent processes.

The behaviour of every object is described as the transitions 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. 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.

A message contains some data and a selector, which is an integer that specifies a task to be done by the receiver of the message. When an object receives a message, the function corresponding to the selector is invoked, with the data in the message as its argument. A function corresponding to a selector is called a "method".

An important feature of objects is that they are single-threaded. This means an object can process only one message at a time. If an object receives a message while it is processing another message, the second message is put into the message queue and processed later.

The typical life cycle of an object can be divided in the next steps:

(1) The object is loaded by the system.

(2) The object waits for a message.

(3) When a message arrives, the object executes the corresponding method. Possibly sends some messages to other objects.

(4) When the method finishes execution, goes to step 2.

Note that this is an infinite-loop: an object can not terminate itself. It persists until the system is deactivated.

2.1.5 The stub.cfg config file

A stub is used to connect an entry point of an object with a member function in a core class. The stub is defined in *xxxStub.cc*, which is automatically generated from *stub.cfg*, by a stub generator (the stubgen2 command). Only one instance of a core class is generated as a global variable in *xxxStub.cc* and every object has its own *stub.cfg* file.

The "Stubgen2" command reads *stub.cfg* and generates intermediate files to connect the methods of an object with the member functions of a core class. *def.h* is one file that is generated by Stubgen2.

The following items are described in *stub.cfg*:

- The number of subjects and the number of observers
- Services used in inter-object communication

The subjects and observers provide the services for inter-object communication. Each service has a unique name, in order to distinguish that service from other services in the system. You have to connect the subject's service to the observer's service by describing both service names in *connect.cfg* (we will see this file in the next point).

The following is a sample of a *stub.cfg* file:

```
ObjectName : SampleClass
NumOfOSubject : 1
NumOfOObserver : 2
Service : "SampleClass.Func1.Data1.S", Control(), Ready()
Service : "SampleClass.Func2.Data2.O", Connect(), Notify1()
Service : "SampleClass.Func3.Data2.O", null, Notify2()
```

The descriptions of each item are:

- **ObjectName**: The core class name.
- **NumOfOSubject**: This is the number of subjects. You must specify at least 1 subject. When you do not need a subject in your program, you should register one dummy subject.
- **NumOfOObserver**: This is the number of observers. You must specify at least 1. When you do not need an observer in your program, you should register one dummy observer.
- **Service**: Here, you specify the communication service for the object. A service corresponding to each subject and observer is described. A service consists of the following items below:

  "(Connection name)", (Member function 1), (Member function 2)

  - **Connection name**: The connection name consists of the following items.

    (Object name). (Subname). (Data name). (Service type)

    - Object Name: You can use any name you like, but this is usually the core class name.
    - Subname: This is a service name and must be unique. Do not use the same subname that other services use.
    - Data name: This is the name corresponding to the data type used in inter-object communication.
    - Service type: S(subject) or O(observer) is specified.
  - **Member function 1:** This member function is called when a connection result is received. You can freely use any names for this function. This function is implemented in the core class. In case you do not need it, you can specify "null" here.
  - **Member function 2:** If this service is for observers, this function is called when a message is received from a subject. If this service is for subjects, this function is called when ASSERT-READY or DEASSERT-READY is received from an observer. You can use any name you like for this function. This function is implemented in the core class. In case you do not need it, you can specify "null" here.

Next, we will explain how messages are sent and received between the subject "a" of object A and the observer "b" of object B, using Figure 5 and the next sample descriptions of the *stub.cfg* file for each object.

```
ObjectName : ObjectA
NumOfOSubject : 1
NumOfOObserver : 1
Service : "ObjectA.SendString.char.S", null, subject_a()
Service : "ObjectA.DummyObserver.DontConnect.O", null, null
```

```
ObjectName : ObjectB
NumOfOSubject : 1
NumOfOObserver : 1
Service : "ObjectB.DummySubject.DontConnect.S", null, null
Service : "ObjectB.ReceiveString.char.O", null, observer_b()
```

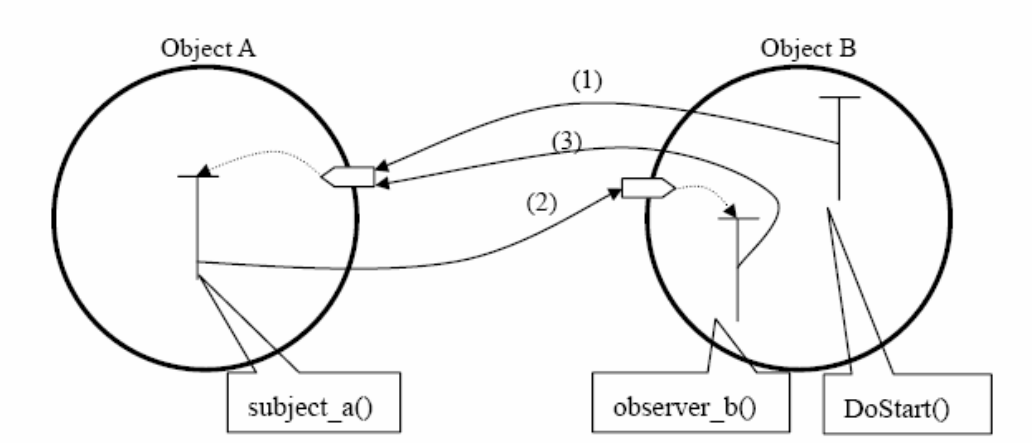

Figure 5: Example of an inter-object communication.

The following are the steps of sending and receiving a message:

- *DoStart()* in object B sends *ASSERT-READY* to the subject in object A. This notification reaches *subject_a()* of the core class in object A.
- *subject_a()* sends a message to Object B. This notification reaches *observer_b()* of the core class in object B.
- When *observer_b()* requests Object A to send the next message, *observer_b()* sends *ASSERT-READY* to Object A.

2.1.6 The connect.cfg config file

The connection between a subject and an object is described in *connect.cfg*. This is a unique file by program and it must be placed in the *OPENR/ MW/CONF/* directory of the memory stick.

An example of a *connect.cfg* file is:

```
Class1.Func1.Data1.S Class2.Func2.Data1.O
```

Each line includes the following items.

"a subject service" (.S) ,"a space", "an observer service (.O)"

The data name in the subject service and the data name in the observer service must be the same.

For example, the connection of the service between subject a and observer b, see figure 5, is described in *connect.cfg* as follows.

ObjectA.SendString.char.S ObjectB.ReceiveString.char.O

2.1.7 The Notify() function

The *Notify(const ONotifyEvent& event)* is a special function that is called when a message arrives in the gate of the object where it is in. It is the only way to retrieve the message that was sent to the object. Therefore, it is the entry point that one must use in order to get the data from a sensor or from function's argument each time that such a message is ready. Its content can be retrieved by casting the variable in which it was copied in (in our case it is the variable event):

DataType dt = (DataType*)event.Data(0);*

At the end of the *Notify()* function, an *AssertReady* must be sent to the subject that sent the message with:

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

2.2 Aibo's camera

2.2.1 Accessing the camera

Sensors and joints are called primitives in Sony's official documentation. In Aibo's design, each primitive can be referred to by using a primitive locator supplied in the Sony's model information document. The primitive locator provides the address of the primitive and the *OPENR::OpenPrimitive* static function convert this adress to an ID. In OPEN-R SDK the type *OprimitiveID* holds ID information. This design was chosen by Sony's developers in order to make objects portable between different Aibo models since the same sensor can have a different index within two different models

2.2.2 Format type of the camera data

OFbkImageVectorData is the data structure that holds image data. It is the type of the data sent from the camera, i.e. the type of the messages sent from the outgoing gate named *FbkImageSensor* of *OVirtualRobotComm* . Actually, a *OFbkImageVectorData* message contains the same picture in different resolutions but all in color that are stored in different layers accessible by their indices. The index of the layer can be one of the following predefined constants: *ofbkimageLAYER_H* (high resolution), *ofbkimageLAYER_M* (medium resolution), *ofbkimageLAYER_L* (low resolution).

Images are in the YCrCb format, which means they are coded using 3 bands: Y luminance, Cr (red component - Y) and Cb (blue component - Y).

The *OFbkImageVectorData* object contains several data and information objects, just as the *OCommandVectorData* object. *OFbkImageVectorData* has a few methods to obtain this data and information like *GetData()* and *GetInfo()*. Of course it is also necessary to know the amount of data objects when using these methods, therefore the *maxNumData* member can be used. Needless to say, in this case the data represents images. Figure 6 shows the structure of the *OFbkImageVectorData* class.

Getting information about an image is done by calling the *GetInfo()* method. This method returns an *OFbkImageInfo* object which can be used to obtain the size and some other information of the image. Getting the corresponding image data can be done by the *GetData()* method. *GetData()* returns a pointer to a byte array that represents the image.

For the programmer it is not necessary to know how an image is stored. To read out the color of a pixel from an image the programmer only needs to use the *OFbkImage* object. *OFbkImage* has methods to easily access image data without having to know the underlying structure. The constructor of this class requires the information and data objects. To retrieve these objects the *GetInfo()* and *GetData()* methods can be used.

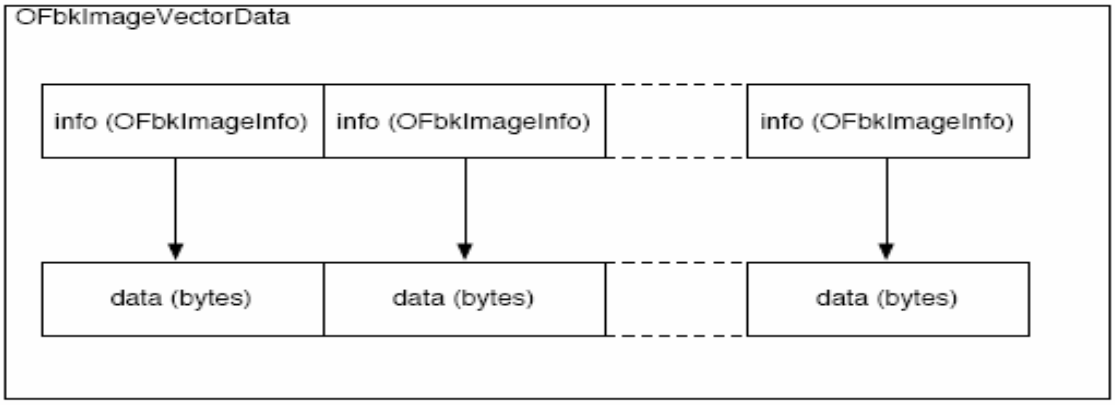

Figure 6. OFbkImageVectorData structure

For an example of the access to the Aibo's camera see the document attached at the end of this report.

2.3 Webots cross-compilation for Aibo robots

As we have mentioned in point 1.2, Webots allows us the use of cross-compilation for Aibo robots, wherein the controller code is cross-compiled to produce a binary executable which then runs on the live robot directly.

There are some test controllers coming with Webots for the two Aibo models placed in the controllers directory of Webots webots/controllers/ (these controllers are called ers7* and ers210*). The controller code is contained in the files named ers7*.c and ers210*.c (the directory and the name of the controller have to be the same in Webots). There are two makefiles inside these directories: Makefile and Makefile.openr. Makefile is necessary for the compilation of the controller for Webots execution and Makefile.openr manage the cross-compilation creating the executable code for Aibo robots. The source code files to be compiled by both makefiles are listed in *Makefile.sources*.

For creating new controllers we will use the existing Aibo base controllers as initial patterns. Below we show the steps for creating the Aibo executable code from a Webots controller (see figure 7):

1. First we should copy to the memory stick an OPEN-R system directory from the OPEN-R SDK containing the system configuration (see figure 8). These files are inside the OPEN-R SDK directory in *OPEN_R/MS/* and there are three posible

configurations: BASIC (without a wireless LAN environment), WLAN (with wireless LAN and without wireless console) and WCONSOLE (with both wireless LAN and console). Each one of these configurations can use a system with memory protection (*memprot* o *nomemprot* directories).

The actual cross-compiler is Sony's OPEN-R cross-compiler included in the OPENR SDK available for download from the official OPEN-R website. In order to take advantage of controller cross-compilation in Webots, the OPEN-R SDK must therefore be installed first.

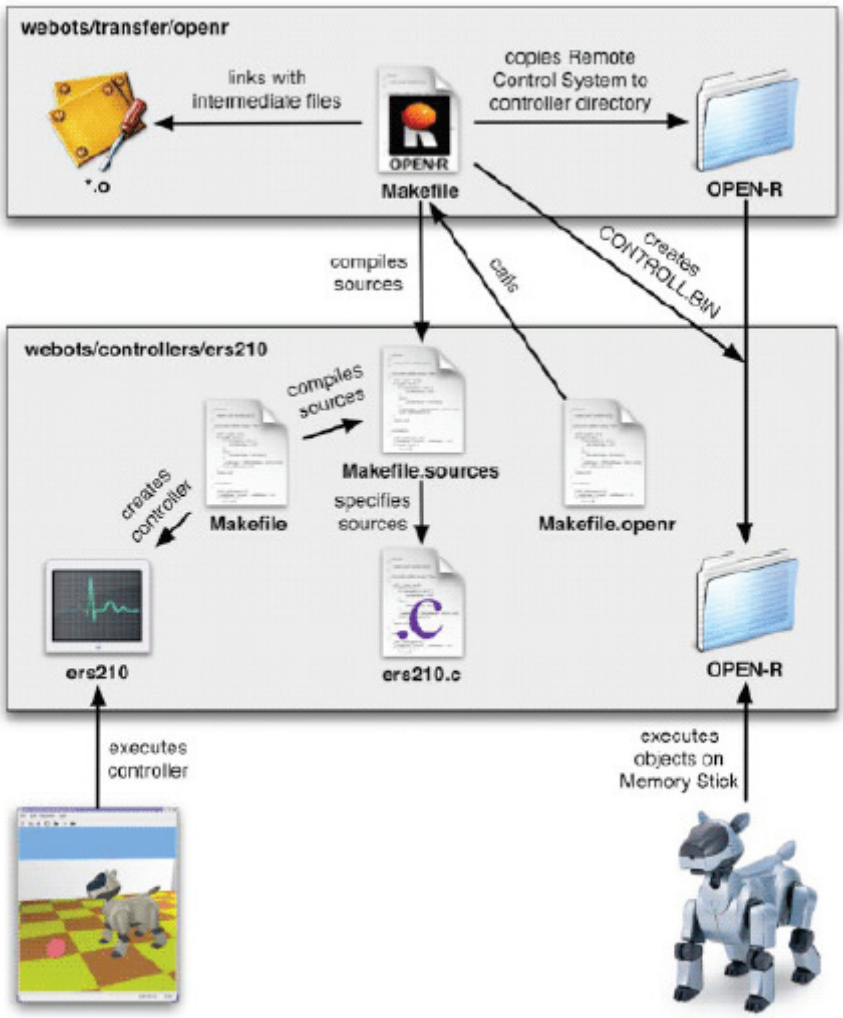

Figure 7. Aibo cross-compilation.

/OPEN-R/	MW/	CONF/		Configuration
		DATA/	E/	
			P/	Content data
		OBJS/		Executable files
	SYSTEM/			System files

Figure 8. OPEN-R directory.

2. Then we should execute the file *Makefile.openr* (make -f Makefile.openr), which calls the files placed in the Webots folder *webots/transfer/openr* (see figure 9). This directory contains intermediate cross-compiled files of the *Controller* OPEN-R wrapper object. *Makefile.openr* compiles all source code files specified in *Makefile.sources* (*.c, *.cc or *.cpp) using Sony's cross-compiler and links them with the already existing files. If there is not yet a directory called OPEN-R/ in the controller directory, a default OPEN-R directory is copied from *webots/transfer/openr*. Calling *make -f Makefile.openr clean* will again remove the OPEN-R/ directory.

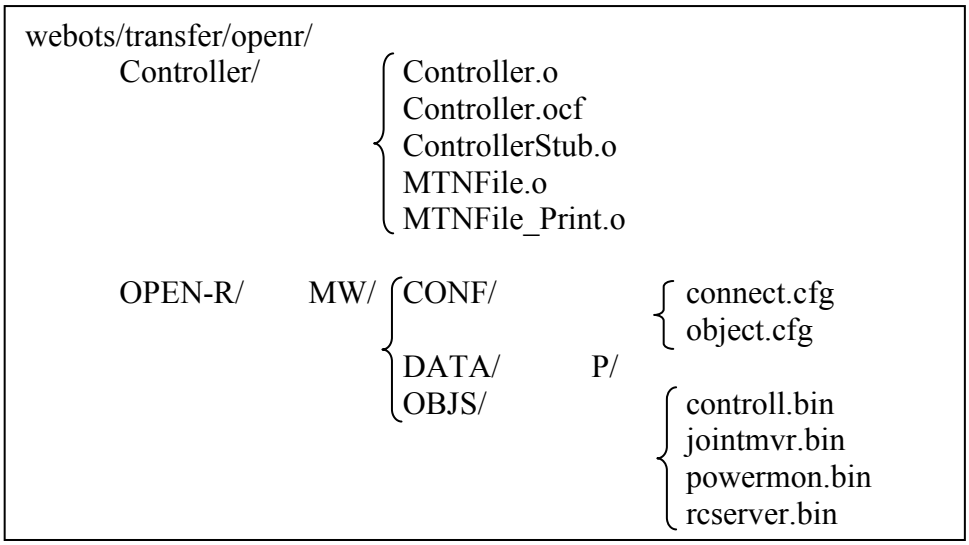

Figure 9. The webots/transfer/openr directory.

3. The controller binary file *CONTROLL.BIN* resulting from the controller cross-compilation, using the wrapper object files placed in *webots/transer/openr/Controller*, is then placed into the *OPEN-R/MW/OBJS/* subdirectory in the *webots/controllers/my_controller* directory. The binary files of the other remote software objects are also located in that target subdirectory. (Initially, the *CONTROLL.BIN* binary code already in place corresponds to a void controller.) Four MTN motion sequence files are already present in *OPEN-R/MW/DATA/P/* and thus do not need to be uploaded separately.
4. The *webots/controllers/my_controller/OPEN-R/* directory may be directly copied to the Memory stick, overwriting the adecuate files of the OPEN-R directory created previously from the OPEN-R SDK. The archives *MTN* that can be used have to be copied in the directory *OPEN-R/MW/DATA/P/* of the Memory Stick.

In order to be able to add the necessary functions for the use of the camera in the OPEN-R wrapper object (*Controller*), we had to modify the source files of the wrapper object, which generates the object files found inside the *webots/transfer/openr/Controller/* folder. In the next chapter we will see how this has been done.

Chapter 3

Adding camera functions to the Webots OPEN-R wrapper object for Aibo robots

In order to be able to extend the wrapper object with the purpose of being able to use the camera, we have to modify the source files of the Webots wrapper object that we will describe in this chapter.

3.1 Controller.h

We have to add to the *Controller* object, in the *Controller* header, the next functions targeted from the Webots API:

```
unsigned short camera_get_width(DeviceTag t);
unsigned short camera_get_height(DeviceTag t);
unsigned char *camera_get_image(DeviceTag t);
float camera_get_fov(DeviceTag t);
```

We will also add the function *Notify* that we use to receive the information from the camera:

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

And the macros used to obtain a colour component from a pixel of the image:

```
#define camera_image_get_red(image,width,x,y)   (image[3*((y)*(width)+(x))])
#define camera_image_get_green(image,width,x,y) (image[3*((y)*(width)+(x))+1])
#define camera_image_get_blue(image,width,x,y)  (image[3*((y)*(width)+(x))+2])
```

We also declare the next private variables:

```
//Array to keep the image in RGB
unsigned char *VectorImage;

//Array to obtain the image from the Aibo camera
OFbkImageVectorData* fbkImageVectorData;

//The index for the vector of pixels
static const int B_PIXEL = 0;
static const int G_PIXEL = 1;
static const int R_PIXEL = 2;
```

3.2 Controller.cc

3.2.1 Notify method

In the *Controler.cc* source file, first we define the function *Notify*:

```
void Controller::Notify (const ONotifyEvent& event) {

        //To receive data from the public area, invoke event.Data(0)
        fbkImageVectorData = (OFbkImageVectorData*)event.Data(0);

        //To request the next data, invoke AssertReady()
        observer[event.ObsIndex()]->AssertReady();
}
```

The last line means that the object is prepared to receive the following event. We will have in *fbkImageVectorData* all the information of the captured image.

3.2.2 camera_enable and camera_disable

These functions do not need code for the real Aibo robot.

```
void camera_enable(DeviceTag t, unsigned short ms) {}
void camera_disable(DeviceTag t) {}
```

3.2.3 camera_get_width and camera_get_height

The function *camera_get_width* returns the width of the image. We use the information of the high resolution image.

```
unsigned short camera_get_width (DeviceTag t) {
        return mySelf->camera_get_width(t);
}

unsigned short Controller::camera_get_width (DeviceTag t) {

        //The image of high resolution
        OFbkImageLayer layer = ofbkimageLAYER_H;

        // Obtaining the information of the camera
        OFbkImageInfo *info = fbkImageVectorData->GetInfo(layer);

        // Obtaining the information of the image
        byte *data = fbkImageVectorData->GetData(layer);

        //Obtaining the image by layers in YCrCb
        OFbkImage yImage(info, data, ofbkimageBAND_Y);
        OFbkImage uImage(info, data, ofbkimageBAND_Cb);
        OFbkImage vImage(info, data, ofbkimageBAND_Cr);

        //Using the function int Width() of the class OFbkImage to obtain the
        //width of the image
        slongword width = yImage.Width();
        return (unsigned short)width;

}
```

The function camera_get_height returns the height of the image. We also use the information of the high resolution image.

```
unsigned short camera_get_height (DeviceTag t) {
        return mySelf->camera_get_height(t);
}

unsigned short Controller::camera_get_height (DeviceTag t) {

        //The image of high resolution
        OFbkImageLayer layer = ofbkimageLAYER_H;

        // Obtaining the information of the camera
        OFbkImageInfo *info = fbkImageVectorData->GetInfo(layer);

        // Obtaining the information of the image
        byte *data = fbkImageVectorData->GetData(layer);

        //Obtaining the image by layers in YCrCb
        OFbkImage yImage(info, data, ofbkimageBAND_Y);
        OFbkImage uImage(info, data, ofbkimageBAND_Cb);
        OFbkImage vImage(info, data, ofbkimageBAND_Cr);

        //Using the function int Height() of the class OFbkImage to obtain the
        //height of the image
        slongword height = yImage.Height();
        return (unsigned short)height;
}
```

3.2.4 camera_get_image

The function *camera_get_image* returns an array with the components of the image. The Aibo color camera uses a YUV format of space of colors. We have to transform the YUV format to RGB for compatibility with Webots. For the conversion, we use the next function, included in the file *ycrcb2rgb.h* that we must include in *Controller.cc*.

```
void YCrCb2RGB(byte y, byte cr, byte cb, byte* r, byte* g, byte* b) {
        double Y, Cr, Cb, R, G, B;

        //Normalization of the data
        sbyte scr = (sbyte)(cr ^ 0x80);
        sbyte scb = (sbyte)(cb ^ 0x80);

        Y  = (double)y / 255.0;     // 0.0 <= Y  <= 1.0
        Cr = (double)scr / 128.0;  // -1.0 <= Cr <  1.0
        Cb = (double)scb / 128.0; // -1.0 <= Cb <  1.0

        //Conversion
        R = 255.0*(Y + Cr);
```

```
        G = 255.0*(Y - 0.51*Cr - 0.19*Cb);
        B = 255.0*(Y + Cb);

        //We verify that the values are between 0 and 255.
        if (R > 255.0) {
            *r = 255;
        } else if (R < 0.0) {
            *r = 0;
        } else {
            *r = (byte)R;
        }

        if (G > 255.0) {
            *g = 255;
        } else if (G < 0.0) {
            *g = 0;
        } else {
            *g = (byte)G;
        }

        if (B > 255.0) {
            *b = 255;
        } else if (B < 0.0) {
            *b = 0;
        } else {
            *b = (byte)B;
        }
}
```

In the function *YCrCb2RGB,* we normalize the data, because the YUV format has a range that goes from -1 to 1 and in RGB the range goes from 0 to 255. Then we do the conversion from YUV to RGB and finally we verify that the final values are between 0 and 255.

In *camera_get_image* we divide the image in YUV components. We convert each component of the image to the corresponding RGB value, and we kept them in order r, g, b in an array called *VectorImage*. This array is defined in the constructor with the dimensions of the image of high resolution multiplied by the three components in which the image is divided. The function returns this array.

```
        unsigned char *camera_get_image(DeviceTag t) {
                return mySelf->camera_get_image(t);
        }

        unsigned char *
        Controller::camera_get_image(DeviceTag t) {
                //The image of high resolution
                OFbkImageLayer layer = ofbkimageLAYER_H;
                byte pixel[3];
                //It obtains the information of the camera
```

```
        OFbkImageInfo *info = fbkImageVectorData->GetInfo(layer);
        //It obtains the information of the image
        byte *data = fbkImageVectorData->GetData(layer);
        //It obtains the image by layers in YCrCb
        OFbkImage yImage(info, data, ofbkimageBAND_Y);
        OFbkImage crImage(info, data, ofbkimageBAND_Cr);
        OFbkImage cbImage(info, data, ofbkimageBAND_Cb);
        //Width ahd height of the image
        slongword w = yImage.Width();
        slongword h = yImage.Height();
        int f = 0;
        for(int y=0;y<h;y++){
                for(int x=0;x<w;x++){
                    //conversion of YCrCb to RGB
                    YCrCb2RGB(yImage.Pixel(x,y), crImage.Pixel(x,y),
                    cbImage.Pixel(x,y),
                    &pixel[R_PIXEL],&pixel[G_PIXEL],&pixel[B_PIXEL]);

                     //It keeps each layers in a position of the vector
                     VectorImage[f]=pixel[R_PIXEL];
                     VectorImage[f+1]=pixel[G_PIXEL];
                     VectorImage[f+2]=pixel[B_PIXEL];

                     f=f+3;
                }
        }
        return VectorImage;
}
```

3.2.5 camera_get_fov

The last function is *camera_get_fov*. In Webots there is only one value for the camera fov value and you can choose that value, but in the real Aibo robot there are two different values, its lens has a opening of 56.7° horizontal and 45.2° vertical.

We return only one value for compatibility with Webots, the height in radians.

```
float camera_get_fov(DeviceTag t){
        return mySelf->camera_get_fov(t);
}

float
Controller::camera_get_fov(DeviceTag t){
        return 0.993092
}
```

3.3 stub.cfg

This file contains the inter-connect objects information and it is necessary to define the subjects and observers of the object.

We have to add the next row to the file *stub.cfg* that we found in the source directory of the *Controller* object:

```
Service : "Controller.Image.OFbkImageVectorData.O", null, Notify()
```

And we have to add one to the *NumOfOObserver*.

3.4 connect.cfg

The *connect.cfg* file is located in the directory */OPEN-R/MW/CONF* of the memory stick. It contains the connections between the subjects and the observers.

We have added the next rows:

```
OvirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S
Controller.Image.OFbkImageVectorData.O
```

References

This report is based on the information that can be found mainly in the following articles:

- *Remote control of the Aibo™ camera from Webots™*. Raphaël Haberer-Proust, Olivier Michel, Auke Jan Ijspeert. 2006 Semester Project. School of Computer & Communication Sciences, Swiss Federal Institute of Technology, Lausanne (EPFL). http://birg.epfl.ch/webdav/site/birg/users/161181/public/report-semester_project.pdf
- *Introduction to the Aibo programming environment*. 2005, Ricardo A. Téllez (r_tellez@ouroboros.org). http://www.ouroboros.org/notes.pdf
- *OPEN-R Essentials*. 2004, Ricardo A. Téllez (r_tellez@ouroboros.org). http://www.ouroboros.org/open-r_v1.0.pdf
- *Manual de Open-R*. Francisco Martín Rico, Rafaela González-Careaga. Universidad Rey Juan Carlos, 28933 Móstoles (Spain). {robotica-profes}@gsyc.escet.urjc.es. http://www2.udec.cl/~clcastro/manual-openr.pdf
- *OPEN-R SDK Programmer's Guide*. 2004, Sony Corporation.
- *Webots User Guide*. 2005, Cyberbotics Ltd.
- *Aibo programming using OPEN-R SDK*. 2003, François Serra and Jean-Christophe Baillie. ENSTA. http://www.ensta.fr/˜baillie

Annex I

Example of a Webots controller for Aibo robots

In this attached document we can see a base code of a Webots controller for Aibo robots where an image is captured and stored on disk. This code can be executed both in Webots and live Aibo robots.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Base Webots controller for Aibo robots.
// Cognition for Robotic Research group (C4R2). Jaume I University.
// 03/2007
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Includes ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <fstream>
#include "../../include/device/robot.h"
#include "../../include/device/servo.h"
#include "../../include/device/led.h"
#include "../../include/device/distance_sensor.h"
#include "../../include/device/touch_sensor.h"
#include "../../include/device/camera.h"
#include "../../include/device/mtn.h"

//Constants ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#define SIMULATION_STEP 16
#define IMAGE_SIZE 208*160
#define MTN_REPLAY 15

// Webots compilation.
//#define MTN_PATH "../../data/mtn/ers7/"
//#define MTN_FILE "../../data/mtn/ers7/wwfwd.mtn"

// Aibo compilation.
#define MTN_PATH "/ms/open-r/mw/data/p/"
#define MTN_FILE "/ms/open-r/mw/data/p/wwfwd.mtn"

// Static variables ////////////////////////////////////////////////////////////////////////////////////////////////////////////
static MTN *mtn;
static DeviceTag camera,
        head_distance_near,
        head_distance_far,
        chest_distance_sensor,
        touch_sensor_fore_l,
        touch_sensor_fore_r,
        touch_sensor_hind_l,
        touch_sensor_hind_r,
        head_pan,
        joint[15];
```

```
// Init function ////////////////////////////////////////////////////////////////////////////////////////////////////////////
static void init(void) {
	camera = robot_get_device("PRM:/r1/c1/c2/c3/i1-FbkImageSensor:F1");
	head_distance_near = robot_get_device("PRM:/r1/c1/c2/c3/p1-Sensor:p1");
	head_distance_far = robot_get_device("PRM:/r1/c1/c2/c3/p2-Sensor:p2");
	chest_distance_sensor = robot_get_device("PRM:/p1-Sensor:p1");
	touch_sensor_fore_l = robot_get_device("PRM:/r2/c1/c2/c3/c4-Sensor:24");
	touch_sensor_hind_l = robot_get_device("PRM:/r3/c1/c2/c3/c4-Sensor:34");
	touch_sensor_fore_r = robot_get_device("PRM:/r4/c1/c2/c3/c4-Sensor:44");
	touch_sensor_hind_r = robot_get_device("PRM:/r5/c1/c2/c3/c4-Sensor:54");

	joint[0] = robot_get_device("PRM:/r2/c1-Joint2:21");		// LFLJ1 joint
	joint[1] = robot_get_device("PRM:/r2/c1/c2-Joint2:22");		// LFLJ2 joint
	joint[2] = robot_get_device("PRM:/r2/c1/c2/c3-Joint2:23");	// LFLJ3 joint
	joint[3] = robot_get_device("PRM:/r3/c1-Joint2:31");		// LRLJ1 joint
	joint[4] = robot_get_device("PRM:/r3/c1/c2-Joint2:32");		// LRLJ2 joint
	joint[5] = robot_get_device("PRM:/r3/c1/c2/c3-Joint2:33");	// LRLJ3 joint
	joint[6] = robot_get_device("PRM:/r4/c1-Joint2:41");		// RFLJ1 joint
	joint[7] = robot_get_device("PRM:/r4/c1/c2-Joint2:42");		// RFLJ2 joint
	joint[8] = robot_get_device("PRM:/r4/c1/c2/c3-Joint2:43");	// RFLJ3 joint
	joint[9] = robot_get_device("PRM:/r5/c1-Joint2:51");		// RRLJ1 joint
	joint[10] = robot_get_device("PRM:/r5/c1/c2-Joint2:52");		// RRLJ2 joint
	joint[11] = robot_get_device("PRM:/r5/c1/c2/c3-Joint2:53");	// RRLJ3 joint
	joint[12] = robot_get_device("PRM:/r1/c1-Joint2:11");		// Neck Tilt1 joint
	joint[13] = robot_get_device("PRM:/r1/c1/c2-Joint2:12");		// Neck Pan joint
	joint[14] = robot_get_device("PRM:/r1/c1/c2/c3-Joint2:13");	// Neck Tilt2 joint

	// Enabling sensors.
	camera_enable(camera,SIMULATION_STEP);
	distance_sensor_enable(head_distance_near,SIMULATION_STEP);
	distance_sensor_enable(head_distance_far,SIMULATION_STEP);
	distance_sensor_enable(chest_distance_sensor,SIMULATION_STEP);
	touch_sensor_enable(touch_sensor_fore_l,SIMULATION_STEP);
	touch_sensor_enable(touch_sensor_hind_l,SIMULATION_STEP);
	touch_sensor_enable(touch_sensor_fore_r,SIMULATION_STEP);
	touch_sensor_enable(touch_sensor_hind_r,SIMULATION_STEP);

	// Setting velocity and acceleration of the joints.
	for (int i = 0; i < 15; i++) {
		servo_set_velocity(joint[i], 2.498);
		servo_set_acceleration(joint[i], 2.498);
		servo_enable_position(joint[i], SIMULATION_STEP);
	}

	// Creating the mtn movement.
	mtn = mtn_new("wwfwd.mtn");
}

// Die function ////////////////////////////////////////////////////////////////////////////////////////////////////////////
static void die(void) {
	// Deleting the mtn data.
	if( mtn ) mtn_delete(mtn);
}
```

```
// Run function ////////////////////////////////////////////////////////////////////////////////////////////////////////////
static int run (int ms) {
	static int loop=0;
	unsigned char *image;
	unsigned char *ychannel, *uchannel, *vchannel;
	unsigned char r,g,b;
	int h,w, aux;
	ofstream is("/ms/open-r/mw/data/p/image.raw");

	ychannel = new unsigned char[IMAGE_SIZE];
	uchannel = new unsigned char[IMAGE_SIZE];
	vchannel = new unsigned char[IMAGE_SIZE];

	if ((mtn_is_over(mtn)) && (loop<MTN_REPLAY)){
		// Play mtn until enough loops
		mtn_play(mtn);
		loop++;
	}

	else if ((mtn_is_over(mtn)) && (loop==MTN_REPLAY)) {
		image = camera_get_image(camera);
		w = camera_get_width(camera);
		h = camera_get_height(camera);
		aux = 0;
		for (int j = 0; j< h; j++) {
			for (int i = 0; i < w; i++) {

			r = camera_image_get_red(image,w,i,j);
			g = camera_image_get_green(image,w,i,j);
			b = camera_image_get_blue(image,w,i,j);
			ychannel[aux] = (unsigned char)(0.299 * r + 0.587 * g + 0.114 * b);
			uchannel[aux] = (unsigned char)(0.492 * (b - ychannel[aux]) + 128);
			vchannel[aux] = (unsigned char)(0.5 * (r -ychannel[aux]) + 128);
			is.put(ychannel[aux]);
			is.put(uchannel[aux]);
			is.put(vchannel[aux]);
			aux++;

			}
		}
	}
	return SIMULATION_STEP;
}

// main function ///////////////////////////////////////////////////////////////////////////////////////////////////////////
int main() {
	robot_live(init);
	robot_die(die);
	robot_run(run);
	return 0;
}
```