Aibo programming using OPEN-R SDK
Tutorial

François Serra
Jean-Christophe Baillie
http://www.ensta.fr/~baillie

June 2003

URBI Notice

We have recently (2004) created and released a new interface to program and control Aibos, called URBI (Universal Robotic Body Interface). URBI is an *interface language* based on a client/server architecture and can work together with C++, Java and Matlab (more languages to come) on any operating system. URBI is robot independent, working with Aibo but also with humanoid robots or with the Webots4 simulator. Changing from the real robot to a simulation is just about changing an IP address.

URBI is considerably simpler to use and understand than OPENR. For example, the ball tracking example described in this tutorial is only 3 lines long in URBI... URBI is simpler to use for beginners but comes with many features for advanced programmers, especially useful for robotic applications: simple access to the motors and sensors, parallel processing of complex script commands, event driven programming, complex motor trajectory, extended mutex policies, integrated behavior description,...

For more details about URBI and to download it: http://www.urbiforge.com

The URBI Team.
www.urbiforge.com

ENSTA

Foreword

The Sony Aibo robot is currently a very interesting (and relatively cheap) plateform to conduct research in Robotics and Artificial Intelligence. Aside the numerous captors and actuators, the most important element is that Aibo is programmable. The Aibo programming language, built on top of C++, is provided by Sony as the OPEN-R SDK. This tutorial is intended to ease the use of this SDK.

The official documentation given by Sony on the `www.aibo.com` web page is not currently covering all aspects of OPEN-R. Some tutorials on the OPEN-R website explain in detail specific things but do not cover other important issues. Besides, the "official" OPEN-R Programming Book is only available in japanese for now.

Probably the best advice would be to study the examples given on the Sony web site. Unfortunately some of these examples, especially the BallTrackingHead example, are provided without any inline comment and require a reverse engineering effort to be understood.

For all these reasons, we decided at ENSTA to start an in depth study of these examples, as a base for a complete and efficient OPEN-R Programming Tutorial.

This document begins by describing the OPEN-R architecture and specificities. To ease understanding, we introduce a graphical formalism to represent OPEN-R programs. This formalism could also be used to help designing programs. Afterwards we go into coding details step by step for several specific actions (moving the joints, getting information from sensors) and we translate the graphical formalism into practical OPEN-R/C++ code. At the end of this tutorial, we added a miscellaneous section containing useful programming information that we found disparate on the web.

This tutorial is supposed to be self contained and bring the novice reader to a reasonable understanding of the OPEN-R architecture and programming, from low level details up to high level design advices. We hope this work can prove useful for the research community.

Jean-Christophe Baillie
François Serra
http://uei.ensta.fr/baillie

Contents

Chapter 1

The general structure of an OPEN-R program

In this chapter we will present the overall structure of an OPEN-R program without going into the coding details. The aim is to understand the philosophy of programming with OPEN-R and the general architecture of the SDK.

1.1 Modular programs

OPEN-R programs are built as a collection of concurrently running OPEN-R objects. An OPEN-R object is implemented using a C++ object but these are two different concepts and should not be confused. In this tutorial *object* will always mean "OPEN-R object" and not "C++ object". Each so-called object runs concurrently with the others and objects are able to communicate with each other by message passing. We will illustrate this structure with an example.

1.1.1 An example: BallTrackingHead

BalltrackingHead is a sample program available on Sony's OPEN-R SDK website[1]. This program has the following behaviour: Aibo first moves his legs and his head to an initial posture and then starts to move his head around looking for the pink ball and tracks it once found. He also plays a sound when he has found or lost the ball in his vision field. The figure 1.1, shows the architecture of the program. Each box represents one object of the BallTrackingHead program (in this particular example, there is one object which has the same name than the program: balltrackinghead). The arrows represent the communication axis between objects. Bold faced objects are internal objects that the programmer do not have to code since they are provided by Sony.

Each object plays a specific role :

- BallTrackingHead: is the main module that coordinates the others. Manages the "seeking" and "tracking" behaviour.
- MovingLegs: moves the legs to the dog's sleeping position.
- MovingHead: moves the head up looking ahead.
- LostFoundSound: handles a sound playing queue.
- OVirtualRobotComm: interfaces the program with Aibo's joints, sensors and camera.
- OVirtualRobotAudioComm: interfaces the program with Aibo's audio device.
- PowerMonitor: monitors the battery state and manages the shutdown action.

[1]http://openr.aibo.com, in the member area.

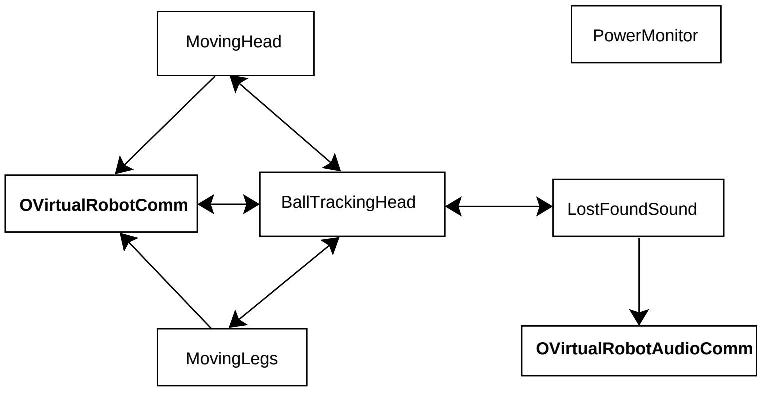

Figure 1.1: The basic structure of BallTrackingHead sample

In this example the BallTrackingHead object is leading the BallTrackingHead program but having such a leader is not always necessary. It depends on the hierarchy the programmer wishes to set between his objects.

Objects can be inter-connected. As we said the arrows symbolize communication axis, which can be seen as a pipe through which messages are sent. The next section will explain further how inter-object communication is done.

1.1.2 Inter-object communication

Synchronization

An OPEN-R object is an independant thread that can communicate with other OPEN-R objects using message passing. In the communication protocol the sender is called the *subject* and the receiver the *observer*. There is a synchronization protocol that can be used to let the observer notify the subject that he is ready to receive and process a message. To do so, the observer sends a special message called *ASSERT_READY (AR)*. This is usually done after he has finished to process the last message. The subject can start a specific action when he receives an AR. Another option for the subject to know that the observer is ready for receiving is simply to ask directly the observer. We will see later how this can be done when it will come to code writing.

Message definition

A message can be a C++ primary type (int, float,...), an array, a structure, a class or a pointer. In the communication protocol, we introduced the notion of communication axis. A communication axis is composed by unidirectional communication channels. Each channel has one fixed subject and one fixed observer. So two channels are required at least to make a bidirectional communication axis. Only one type of message can go through a channel. Thus it is necessary to have two channels to send messages of two different types. The figure 1.2 shows an example of bidirectional communication axis able to carry messages of type A and B in one way and messages of type C in the other way.

The AR message is specific to each channel. If an object is an observer for two or more channels sending an AR message through one channel will not send it in the others.

In the OPEN-R SDK, only channels exist, there is no implementation of the notion of communication axis, which has been presented for sake of clarity in the previous section. In the following, we will always use the channels detail level.

1.1.3 Why modular programs ?

The main advantages of a modular program are:

- Parallel processing.

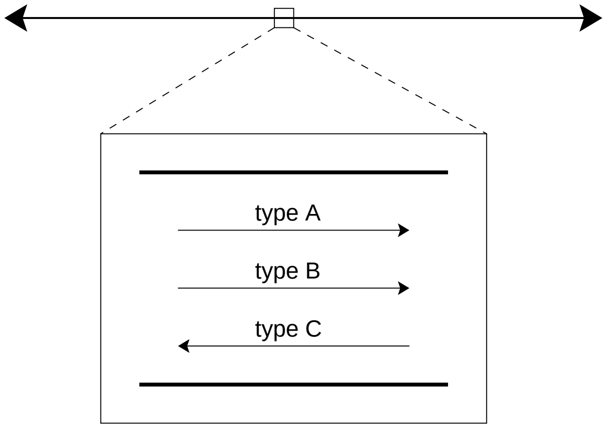

Figure 1.2: An example of a bidirectional channel.

- Clarity of the design: each object handles a set of specific behaviours. For example an object could handle all the head behaviours like “looking ahead” , “looking down” and “swinging the head” while an other could handle the legs behaviours like “raising to standing position”, “walking forward” and “stop”.
- Easiness of reusing pre-existing objects: they do not have to be compiled again since they communicate with message passing.

1.2 Design of an object

1.2.1 Finite states automatons

All examples provided by Sony use a formalism close to finite state automatons to describe the working cycle of an object. Objects have different states and at least the starting state `IDLE` which is a no-operation state and is essential for the system coherence.

As a general design rule, an object cannot be in two different states at the same time.

The automaton changes from a state to another using transitions. A transition is activated by a message and can have multiple paths leading to others states in a tree-like way. Each branch of the tree have a condition. Conditions of the branches descended from the same node must be exclusives (otherwise the automaton could reach two different states at the same time.) The automaton gets over a transition when this transition has been activated by the proper message and the conditions of the path leading to the target state have been satisfied. There is a particular case: the automaton always go trough a transition which have a single path and a condition equal to `1`.

1.2.2 The LostFoundSound object from the BallTrackingHead sample

The aim of this section is to present a graphic representation of the formalism presented previously. We will take the example of the *LostFoundSound* object which appears in the figure 1.3 and is part of the *BallTrackingHead* program.

The different states of the object are represented by circled text. As a convention, the `IDLE` state is always the beginning state.

Messages are represented by dotted arrows. They come in and get out the object through *gates*. The arrow-like shapes representing these gates indicate if they are incoming or outgoing gates. Next to the gates, three labels provide additional information: the first one (bold faced) is the name of the gate (which is usually the name of the observer) and the second (bold faced and italic shape) is the type of the message exchanged. When it exists the third line is the name of a function. In the case of an incoming gate this function will be called each time a message is received, in the case of an outgoing message the function will be called each time an AR is received. This is a basic event driven kind of programming, which is a classic way of handling action-reaction binding in an object oriented framework.

The transition between two states, which is represented by a full line arrow with a black square in the middle, has two requirements:

- The transistion must be activated by an event (receiving a message or an AR)
- The condition on top of the black square must be satisfied.

Going through the transition can possibly trigger one or two of the following actions :

- Process the code which is under the black square.
- Send an outgoing message (represented by a dotted arrow starting from a black dot located on the transition arrow, after the black box).

As we said already, there can be multiple paths for one transition and each of these paths has one or several mutually exclusive conditions attached to.

The box labelled `OVRAC AR` represents an AR message incoming from the *OVirtualRobotAudioComm* (short: OVRAC) object indicating that it's ready to receive. Since the AR messages often activate several transitions in the same object they have been represented by these small labelled boxes to prevent overloading of the diagrams.

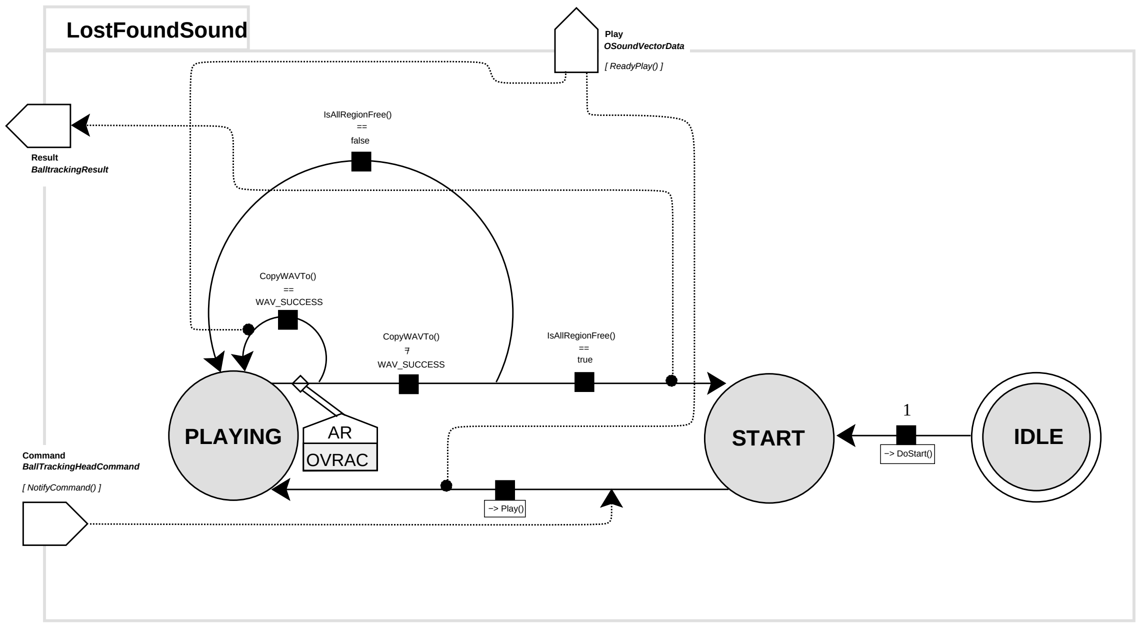

Figure 1.3: The LostFoundSound object from the BallTrackingHead sample

We can now explain step by step the working of the LostFoundSound object :

1. the object is in the `IDLE` state and changes to the `START` state immediatly calling the `DoStart()` function (we will explain what this function does in section 2.2).
2. the object waits for an incoming message to move to the `PLAYING` state.
3. the object remains in the `PLAYING` state each time a *OVirtualRobotAudioComm* AR messages arrives, until the two conditions on the path leading to the `START` state are satisfied (the object returns to step 1). If the first condition is not satisfyed, the paths leading back to the `PLAYING` state will trigger a message through the "play" gate.

Looking close to the working cycle of this simple objects shows that, as it will often be the case, the "heart beat" of the object is given by the incoming AR/OVRAC messages. Without these messages, the object would remain inerte in the `PLAYING` state.

1.3 The full BallTrackingHead example

In the previous sections we have described the basic bricks required to build a complete OPEN-R program and seen in details how an object is designed. Now we will see how all this works when they are combined. This section entirely rely on the BallTrackingHead sample, but prior reading of the source code is not necessary.

1.3.1 Overview

Most Sony example programs begin by bringing Aibo to a standing or sleeping position. In the "BallTrackingHead" program, this action is performed by the objects *MovingHead* and *MovingLegs*. The object *BallTrackingHead* manages the whole program and handles the “searching” and “tracking” behaviours. *LostFoundSound* is a sound playing queue manager, *OVirtualRobotComm* and *OVirtualRobotAudioComm* are special objects which interface with the hardware (*cf.* section 2.1.1)

Figure 1.4 is the diagram of the whole program including every objects and connections. Diagrams of objects alone, with a higher level of detail, are available in the next section. Nevertheless the whole program could nearly be completely understood with this global diagram.

One of the important things to look at when one tries to understand the global diagram is the set of dotted lines representing the message channels and making a kind of “wiring plan” between the objects.

1.3.2 The different objects

BallTrackingHead

The figure 1.5 shows the object *BallTrackingHead.* This object moves directly to the `START` state calling `DoStart()` (the aim of this function will be explained in section 2.2) and sending two messages, one for the "MovingHead" object and one for the "MovingLegs" objects. The purpose of these messages is to put Aibo’s legs and head in the staring position. Once in the `START` state, the BallTrackingHead object can either go to the `HEAD ZERO POS` state (which means that Aibo’s head has reached the desired position but not the legs) or to the `LEGS SLEEPING` state (which means that Aibo’s legs have reached the desired position but not the head), which depends on which of *MovingHead* or *MovingLegs* replied first and activated the corresponding transition to their `IDLE` state. Then the slowest object replies and *BallTrackingHead*’s state changes to `SEARCHING BALL` and the function `SearchBall()` is called, plus a message is sent. This set of four states (`START`, `HEAD ZERO POS`,`LEGS SLEEPING`, `SEARCHING BALL`) can be seen as a synchronization unit between the "MovingHead", "MovingLegs" and "BallTrackingHead" objects.

The transition going from the `SEARCHING BALL` state to the `TRACKING BALL` state is activated when a message is received from the *Image* incoming gate. The type of data is OFbkImageVectorData which is a type of image. Then the transition has a branchement: if the color frequency (here, the pink color) of the image is greater than the threshold `B_THR`, the ball was present in the image and the counter `found` is incremented, else the counter `found` is reset to zero, commands for swinging the head are sent and the object stays in the `SEARCHING BALL` state. When the counter `found` reaches the found threshold `F_THR` (which means the ball has been found in `F_THR` consecutive images) the object gives *LostFoundSound* the order to play a sound, sends a message to move the head toward the ball and moves to the `TRACKING BALL` state. The two thresholds `B_THR` and `F_THR` produce an hysteresis kind of transition. In fact the ball has to be found in `F_THR` consecutive images for the robot to consider the ball as really present in the view of the robot. The hysteresis phenomenon compensates the noise effects in the image.

Once the object is in the `TRACKING BALL` state, a similar behaviour happends, much like in the `SEARCHING BALL` state. In fact *BallTrackingHead* swaps between the searching and tracking states as it finds the ball or loses it.

The state named `*` is a wildcard state meaning "all states".

Here messages coming from *Sensor* or *LostFoundSound* gates do not actually change any state but only perform actions (visible only in the detailed diagram of the BallTrackingHead object, figure 1.5).

LostFoundSound

The figure 1.3 shows the diagram of the *LostFoundSound*. It has one incoming gate and two outgoing ones. As *LostFoundSound* starts in the `IDLE` state it moves to the `START` one where it waits for a message incoming from the *Command* gate in order to move to the `PLAYING` state, calls the `play()` function (which initializes the playing) and sends a message. The *ASSERT_READY* message activates the transition starting from the `PLAYING` state: when the observer is ready *LostFoundSound* copies some WAV information in the shared memory (function `CopyWAVTo()`) and sends message until there is anything to play and the shared memory has been released. Then *LostFoundSound* returns to the `START` state and waits until it receives a new message. Section 4.2 gives a much more detailed description on how to play sound with Aibo, and this is the method used here.

MovingHead

The aim of *MovingHead* is to bring Aibo's head to zero position. This object is a useful example because it shows the basic steps for initializing and moving joints. However, those mechanisms will only be outlined here, since further detailed explanations follow.

The object starts in the `IDLE` state but unlike the previous ones, it stays in this state. It's only when an incoming message arrives from the *command* gate that the object starts moving to an other state (see the `*` state). First if the *Move* gate (*OVirtualRobotComm*, observer) is ready, the function `AdjustDiffJointValue` is called, a message is sent and the object moves to the `ADJ DIFF JOINT VALUE` state. If the observer was not ready, *MovingHead* moves to the `START` state and waits for it. Once again, when *OVirtualRobotComm* is ready (AR/OVRAC received), the function `AdjustDiffJointValue` is called, a message is sent and the object moves to the `ADJ DIFF JOINT VALUE` state.

The aim of the `AdjustDiffJointValue` function is to calibrate the difference between the joints sensors and motors: the function reads the position of each joint first. This position can differ from the actual position of the joint because of uncalibrated sensor values. Then, a command is sent to each joint to actually reach the previously read position, making sure that both read and real positions will be the same.

Once this job is done, the object set the PID gains for each joint (tech. details on PIDs later). `SetJointGain()` does this PID setting when the object changes to the `MOVING TO ZERO POS` state. Then the object makes the head move to the zero position. This is done in `ZP_MAX_COUNTER` equal steps because of a inner mechanisms used by Sony to control the speed of motor movements. At each step the object sends commands to *OVirtualRobotComm* using a buffering method explained in the section 4.1.4. This will all be detailed later on.

Then, after the `ZP_MAX_COUNTER` steps, the object returns to the `IDLE` state.

There is an isolated state `SWING HEAD` which is a forgotten remains from the other program sample called "MovingHead" on top of which BallTrackingHead was built. It should be ignored.

One important thing here is to notice that an object have to do initialisation sequences before moving Aibo's joints (the `AdjustDiffJointValue`'s business).

MovingLegs

MovingLegs is nearly the same as the *MovingHead* object except that there is one more step after having moved Aibo's legs to broadbase (as if the dog was stretching): the legs moves to a sleeping position.

The detailed instructions on how to move the robot to a given position are all stored in an array of joint's values that are associated to "stand", "sleep", "rest",... Some programs are available that record the set of all Aibo's joints values in a given position (set by hand, for example) in order to bring the robot back to this position later. Here, the same method is used to move the robot to the "broadbase" position.

1.3.3 The progress of the program

When Aibo boots, the objects of the BallTrackingHead program are loaded by the operating system and are in the `IDLE` state. The objects *BallTrackingHead* and *lostFoundSound* move directly to their `START` state. At this time *BalltrackingHead* has sent a message to *MovingHead* and *MovingLegs*. Each of them starts and Aibo moves the head to look ahead while his legs move to the sleeping position. When they have finished each one sends a message to *BallTrackingHead* which is waiting for their reply. Then *BallTrackingHead* moves to the `SEARCHING BALL` state. The behaviour has already been described previously. When the ball is found or lost *BallTrackingHead* sends a message to *LostFoundSound*. If the latter is in the `START` state the sound is played as soon as possible. A result message is sent back to *BallTrackingHead* when *LostFoundSound* returns to the `START` state. This message does not affect the *BallTrackingHead* state. The BallTrackingHead program keeps searching and tracking the ball until the power is shut down.

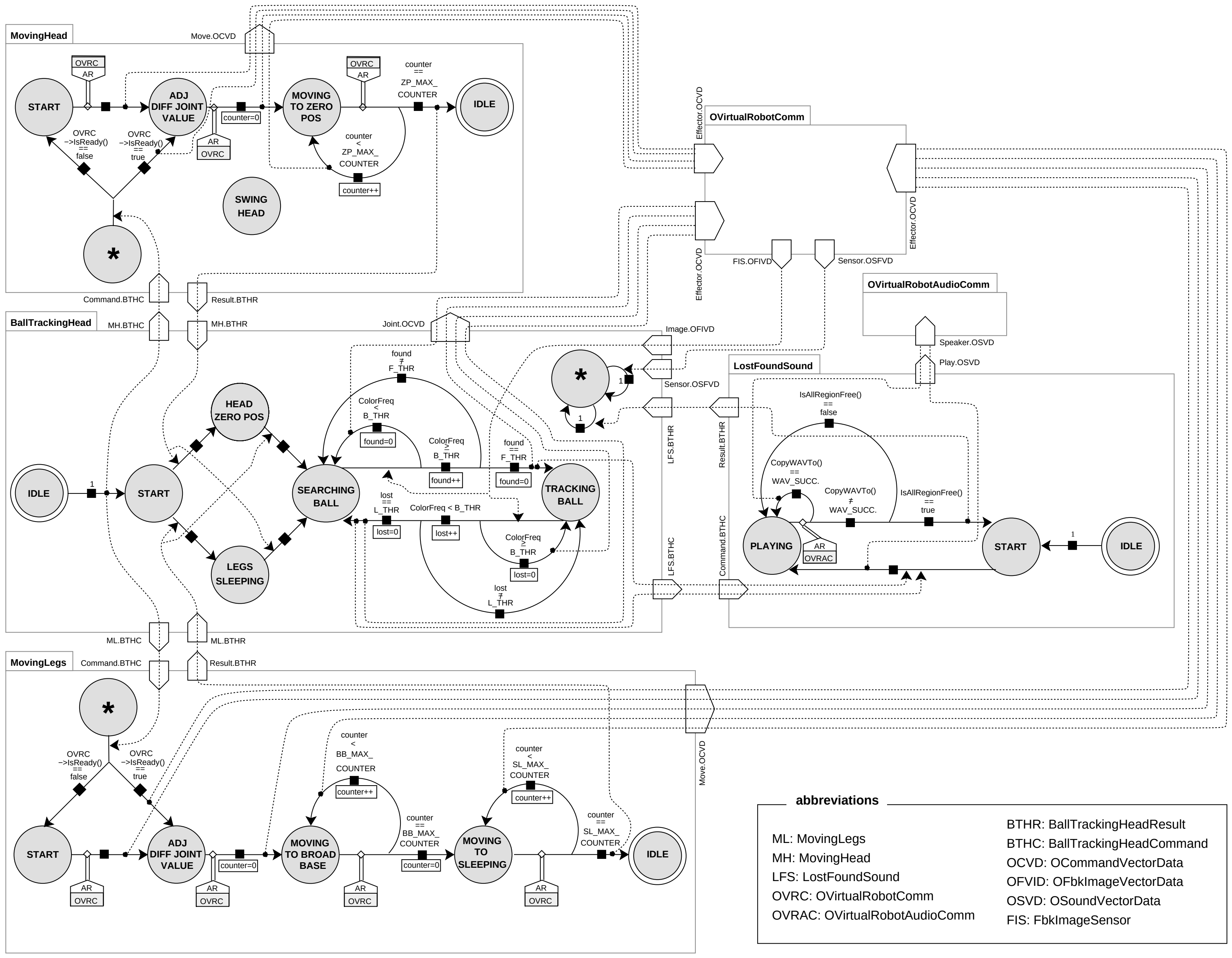

Figure 1.4: The entire BallTrackingHead sample diagram

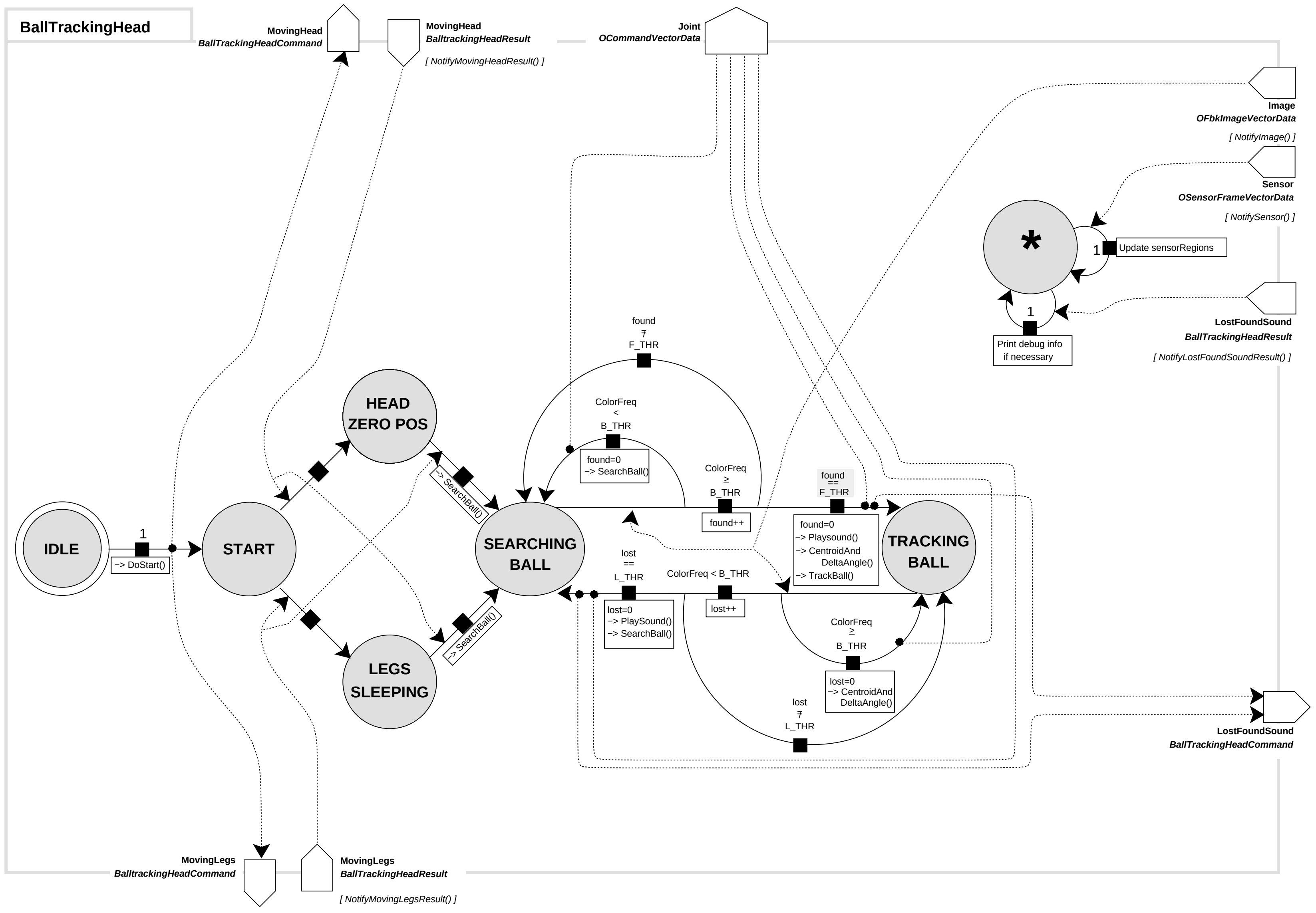

Figure 1.5: The BallTrackingHead object

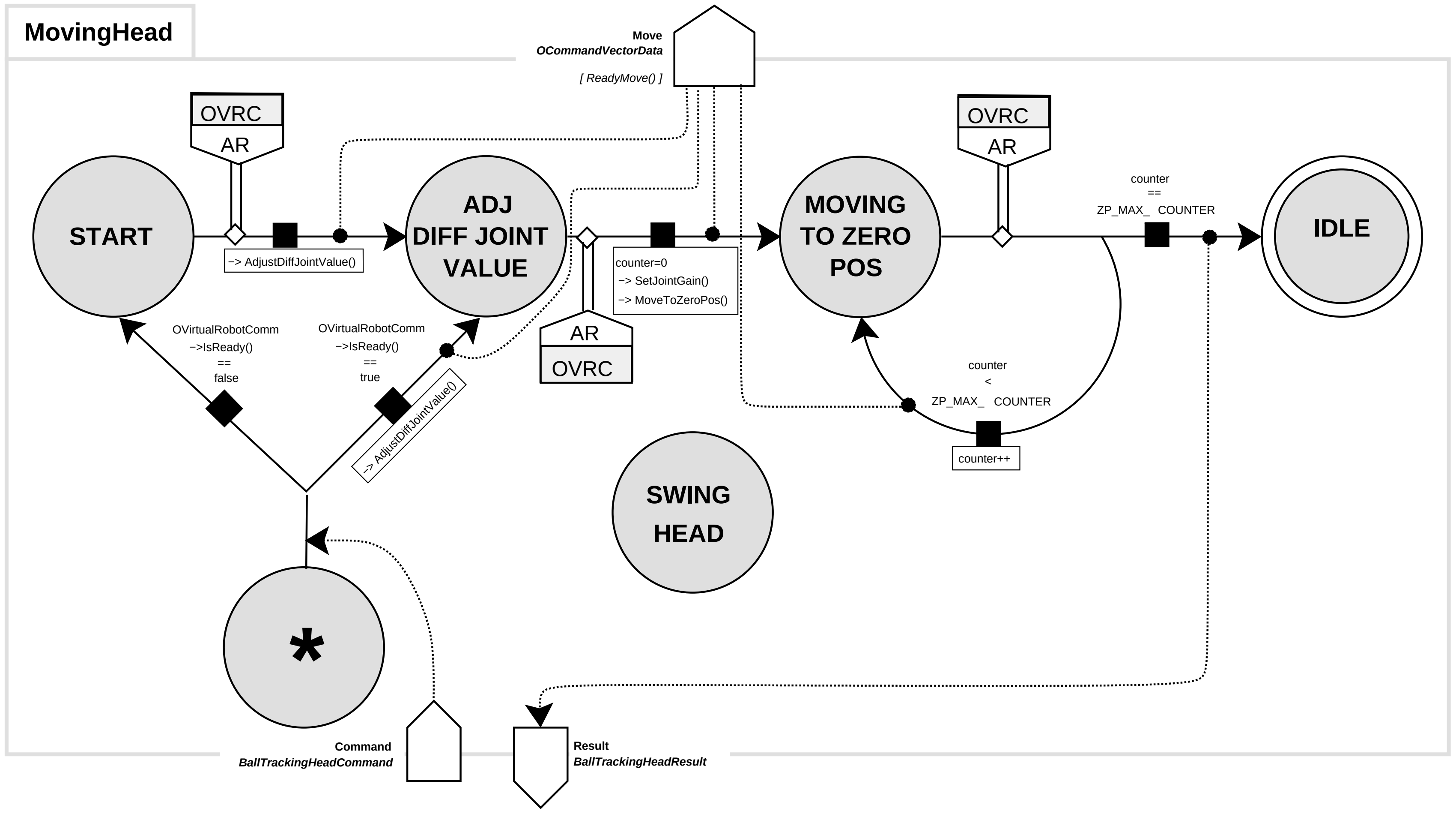

Figure 1.6: The MovingHead object

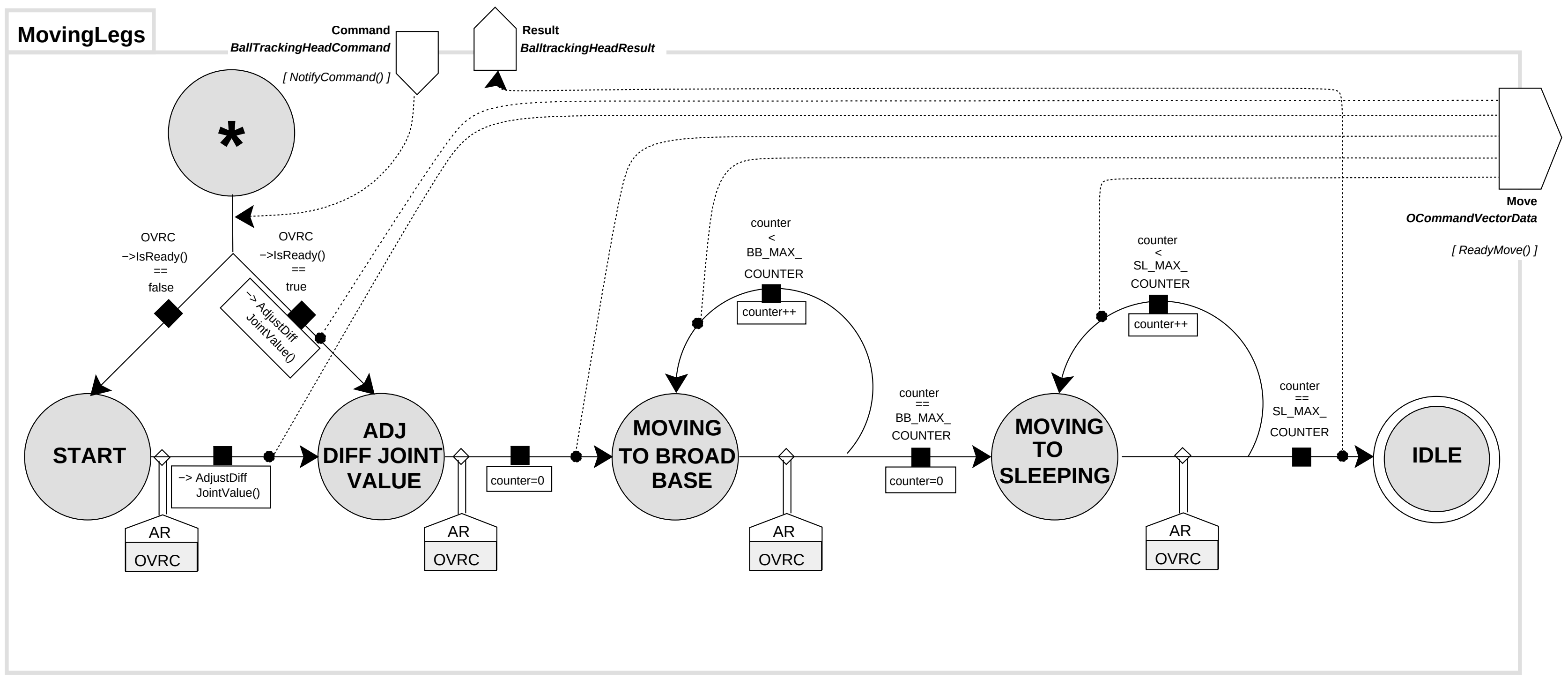

Figure 1.7: The MovingLegs object

Chapter 2

Implementation with the OPEN-R SDK

In the previous chapter we saw the logical structure of an OPEN-R program. In the current chapter, we will study the specificities of the OPEN-R SDK and how to translate this logical structure into C++.

First we need some basic knowledge on how Aibo is working. Then we will explain the skeleton of the C++ class underlying an OPEN-R object. This chapter will end with how to set up the gates of an object and how to configure the inter-connection between the objects.

2.1 Basic knowledge

2.1.1 Objects that interface with the hardware

We have already mentioned that the BallTrackingHead program contains two specific objects provided by Sony: *OVirtualRobotComm* and *OVirtualRobotAudioComm*. These objects are the interface with Aibo's hardware. *OVirtualRobotComm* interfaces with the dog's joints, sensors, LEDs and camera. *OVirtualRobotAudioComm* interfaces with Aibo's audio device (playing or recording sound).

Like any other OPEN-R objects, *OVirtualRobotComm* and *OVirtualRobotAudioComm* communicate with the other objects by message passing. The following types are used in messages to communicate with *OVirtualRobotComm*:

- OCommandVectorData: data structure that holds joint and LED commands.
- OSensorVectorData: data structure that holds sensor information.
- OFbkImageVectorData: data structure that holds image data.

The following types are used to communicate with *OVirtualRobotAudioComm*:

- OSoundVectorData: data strucutre that holds sound data (input and output).

All theses data structures will be explained in the next chapters.

The objects that interface with Aibo's hardware have the same behaviour than all OPEN-R objects: they can be subjects and/or observers. When they are observers they send AR message when they are ready to receive a new message. *OVirtualRobotComm* is an observer receiving messages of the OCommandVectorData type, and has two gates sending messages of OSensorVectorData and OFbkImageVectorData types. Sound can be played or recorded by sending and receiving messages of OSoundVectorData type to *OVirtualRobotAudioComm*.

2.1.2 Frames

Time in Aibo's hardware is discrete. The base unit of the time is a frame, which represents 8 ms. An incrementing counter tags frames so they have a unique number associated, between 0 and oframeMAX_NUMBER (equals to 0x0ffffff0). When the counter reaches oframeMAX_NUMBER it is reset to 0.

Interface objects give information from the sensor by block of n frames, so it shows the history of the sensor ($0 < n \leq$ osensorMAX_FRAMES $= 16$). Those n frames are necessary contiguous in time but not necessarily synchronized with the current frame of the system. For example if the current frame has the number 21, the frames of the block might have the numbers 16, 17 and 18, which means that we could access to the state of the sensor 40, 32 and 24 ms ago. However, at the same time, getting the values from an other sensor, we might get only two frames, starting at frame 14. So, the numbering of frames is very important and allows the programmer to synchronize sensors information by searching matching frame numbers.

As an example, some joint sensors may have a refresh rate which is quicker than the camera refresh rate, so the *OVirtualRobotComm* may have sent two or three times sensors information while it may have sent only one image. Finding the sensors information and image matching frame number gives a "snapshot" of Aibo's posture when the image was taken.

Commands are also sent to Aibo using a block structure. A set of commands is sent for each of the next n frames (where $0 < n \leq$ ocommandMAX_FRAMES $= 16$.) Since motors are commanded by a set of positions to reach, subdividing the desired path into points to reach gives us somehow a control on the velocity of the motors. For example if a motor is in the 0° postion and the programmer asks it to be in the +0.8° and +0.16° position the two next frames, the motor will move from the 0° postion to the +0.16° position with an angular speed of +0.1°/ms.

An other way of implementing the control of motors' velocity would have been to take the starting and the final position plus the desired velocity as input data, what would have induced the duration of the movement. For example a movement from the position p_1 to the position p_2 at a speed of v lasts $\frac{p_2 - p_1}{v}$ seconds. The programming technique used in OPEN-R is more complicated than this but on the other hand, it makes it possible to move joints with non-linear speeds in an easy way. It is easy to create functions that take the starting and the ending position as inputs and calculate the intermediate points to reach. Such functions are indeed used in the BallTrackingHead example.

2.2 The basic C++ class layer of an OPEN-R object

An OPEN-R object is compiled from a C++ class. This class must inheritate from the OObject base class. The constructor of the class should set the state of the object to IDLE, if you want to comply with the Sony's examples standards.

On top of that four virtual functions must be implemented:

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

Here is **OObject.h**

```
//
// OObject.h
//
// Copyright 1997,1998,1999,2000 Sony Corporation
//

#ifndef _OObject_h_DEFINED
#define _OObject_h_DEFINED

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

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

class OObjectManagerProxy;
class OServiceManagerProxy;

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

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

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

protected:
   OID   myOID_;
   OStatus RegisterServiceEntry(const OServiceEntry& entry,const char* name);

private:
   static OObjectManagerProxy    objectManager_;
   static OServiceManagerProxy   serviceManager_;

   // These method are forbidden.
   OObject(const OObject&);
   OObject& operator=(const OObject&);
};

#endif /* _OObject_h_DEFINED */
```

DoInit()

DoInit() is called when the object is loaded by the system. It initializes the gates and registers subjects and observers the object will communicate with. Macros are predefined in the OPEN-R SDK and simplify the description of the working of DoInit().

The skeleton of DoInit() is:

```
OStatus MyObjectClass::DoInit( const OSystemEvent& event )
{
    NEW_ALL_SUBJECT_AND_OBSERVER;
    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;

    /* here the programmer can add his own code */

    return oSUCCESS;
}
```

DoStart()

DoStart() is called after DoInit() is executed in all objects. Here the object usually sends an AR message to all its observers. In the Sony's examples, if the object has to move by himself from the IDLE state to an other, it does it here. There are also predefined macros that simplify the code:

```
OStatus MyExampleClass::DoStart( const OSystemEvent& event )
{
    ENABLE_ALL_SUBJECT;

        /* the programmer can add his code here.
         * For example the object can move by himself to
         * a state
       */

        myExampleClassState = START;

    ASSERT_READY_TO_ALL_OBSERVER;
```

```
        /* code can also be added here */

    return oSUCCESS;
}
```

Sometimes the object does not need to send an AR mesage to all its observers. In that case the programmer sends AR messages manually (that is explained in the section 2.3.3) and does not use the `ASSERT_READY_TO_ALL_OBSERVER` macro.

DoStop()

`DoStop()` is called at shutdown of the system. With the Sony's conventions, the object must move by himself to the `IDLE` state. The function stops the outgoing gates and sends a `DEASSERT_READY` message to all his observers. `DEASSERT_READY` means that the object cannot receive a message anymore. OPEN-R SDK once again provides macros:

```
OStatus MySampleClass::DoStop( const OSystemEvent& event )
{
    myExampleClassState = IDLE;

        /* code can be added here */

    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;

        /* and here */

    return oSUCCESS;
}
```

DoDestroy()

`DoDestroy()` is called at shutdown after `DoStop()` has been called on all objects. Usually this function remains as is:

```
OStatus MyObjectSample::DoDestroy( const OSystemEvent& event )
{
    DELETE_ALL_SUBJECT_AND_OBSERVER;
    return oSUCCESS;
}
```

The complete skeleton of the C++ class

MySampleClass.h:

```
#ifndef MySampleClass_h_DEFINED
#define MySampleClass_h_DEFINED

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

/* The different states of the object : */

enum MySampleClassDifferentStates {
     IDLE,
     /* add here the different states of the object */
};

class MySampleClass : public OObject {
public:

     MySampleClas();
     virtual ~MySampleClas() {}

     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);

private:

         MySampleClassDifferentStates mySampleClassState;
};

#endif // MySampleClass_h_DEFINED
```

During the compilation the file `def.h` is created by the OPENR-SDK tool *stubgen2*. `numOfSubject` and `numOfObserver` are defined in this file. **MySampleClass.cc:**

```
MySampleClass::MySampleClass()
{
         mySampleClassState = IDLE;
}

OStatus MyObjectClass::DoInit( const OSystemEvent& event )
{
     NEW_ALL_SUBJECT_AND_OBSERVER;
     REGISTER_ALL_ENTRY;
     SET_ALL_READY_AND_NOTIFY_ENTRY;

     /* here the programmer can add his own code */

     return oSUCCESS;
}

OStatus MyExampleClass::DoStart( const OSystemEvent& event )
{
     ENABLE_ALL_SUBJECT;

         /* the programmer can add his code here.
          * For example the object can move by himself to
          * a state
        */

         myExampleClassState = START;

     ASSERT_READY_TO_ALL_OBSERVER;

         /* code can also be added here */

     return oSUCCESS;
}

OStatus MySampleClass::DoStop( const OSystemEvent& event )
{
    myExampleClassState = IDLE;

        /* code can be added here */

    DISABLE_ALL_SUBJECT;
    DEASSERT_READY_TO_ALL_OBSERVER;

        /* and here */

    return oSUCCESS;
}

OStatus MyObjectSample::DoDestroy( const OSystemEvent& event )
{
    DELETE_ALL_SUBJECT_AND_OBSERVER;
    return oSUCCESS;
}
```

2.3 Setting and using Inter-object communication

The next section explains how to set-up the inter-object configuration files. It also describes in concrete terms how to send a message, an AR message, how to know if an observer is ready and what to do when receiving a message.

2.3.1 The file stub.cfg

Subjects and Observers are *services* in the OPEN-R SDK terminology. Services of an object are declared in one separate file specific to each object: `stub.cfg`. This file is read by the OPEN-R SDK tool named *Stubgen2* just before calling *gcc*. This tool creates `def.h`, `NameOfObjectStub.cc` and `NameOfObjectStub.h` files which define several macros and constants.

`stub.cfg` begins by a line describing the name of the object. The next two lines declare how many subjects and observers the object have. Then each service is described on a line. Here is the example of the `stub.cfg` of the *LostFoundSound* object:

```
ObjectName : LostFoundSound
NumOfOSubject    : 2
NumOfOObserver   : 1
Service : "LostFoundSound.Play.OSoundVectorData.S", null, ReadyPlay()
Service : "LostFoundSound.Command.BallTrackingHeadCommand.O", null, NotifyCommand()
Service : "LostFoundSound.Result.BallTrackingHeadResult.S", null, null
```

Actually, each line describing a service holds the same information as gate labels in the graphic formalism: in `LostFoundSound.Play.OSoundVectorData.S`, `LostFoundSound` is the name of the current object, `Play` the name of the gate, `OSoundVectorData` the type of the message exchanged and `S` means it is a subject (outgoing gate). `O` will have meant that it was an observer. The `O` and the `S` are represented by the orientation of the gates in the diagram.

The last two fields are the name of two functions: the first one is called when a connection result is received (in almost every cases this function is unusefull so it is set to “null”) and the second one is called when an AR or a message is received. Figure 2.1 shows an example of a gate in the graphical formalism and the corresponding line in the `stub.cfg` file.

2.3.2 The file connect.cfg

In order to interconnect objects, we must assign each subject to one observer and each observer to a subject. The config file doing this is the `connect.cfg` file which should be in the `OPEN-R/MW/CONF/` directory of the Aibo Programming Memory Stick. When Aibo boots up, the system loads the objects and interconnect them using `connect.cfg`. Here is `connect.cfg` for the BallTrackingHead program:

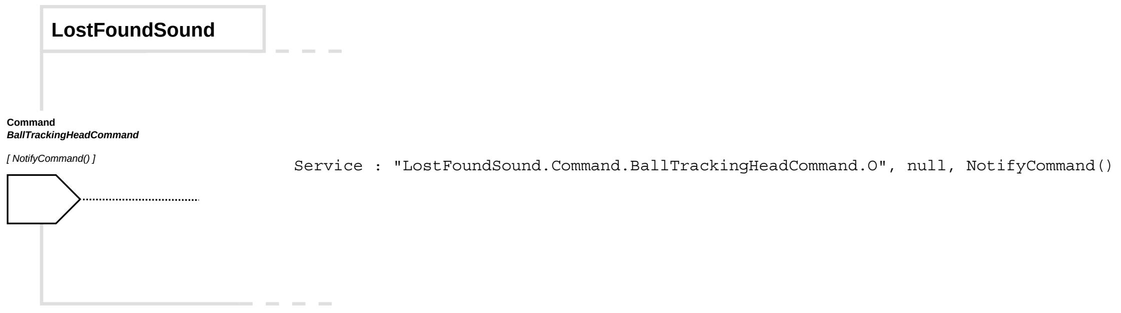
LostFoundSound
Command
BallTrackingHeadCommand
[NotifyCommand()]
Service : "LostFoundSound.Command.BallTrackingHeadCommand.O", null, NotifyCommand()

Figure 2.1: Equivalence between the graphical formalism and the `stub.cfg` file

```
#
# ballTrackingHead <-> movingHead2
#
BallTrackingHead.MovingHead.BallTrackingHeadCommand.S MovingHead.Command.BallTrackingHead
Command.O
MovingHead.Result.BallTrackingHeadResult.S BallTrackingHead.MovingHead.BallTrackingHeadRe
sult.O


#
# ballTrackingHead <-> movingLegs2
#
BallTrackingHead.MovingLegs.BallTrackingHeadCommand.S MovingLegs.Command.BallTrackingHead
Command.O
MovingLegs.Result.BallTrackingHeadResult.S BallTrackingHead.MovingLegs.BallTrackingHeadRe
sult.O

#
# ballTrackingHead <-> lostFoundSound
#
BallTrackingHead.LostFoundSound.BallTrackingHeadCommand.S LostFoundSound.Command.BallTrac
kingHeadCommand.O
LostFoundSound.Result.BallTrackingHeadResult.S BallTrackingHead.LostFoundSound.BallTracki
ngHeadResult.O

#
# ballTrackingHead <-> ovirtualRobotComm
#
OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S BallTrackingHead.Image.OFbkImageVe
ctorData.O
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S BallTrackingHead.Sensor.OSensorFrameVec
torData.O
BallTrackingHead.Joint.OCommandVectorData.S OVirtualRobotComm.Effector.OCommandVectorData
.O

#
# movingHead2, movingLegs2 -> ovirtualRobotComm
#
MovingHead.Move.OCommandVectorData.S OVirtualRobotComm.Effector.OCommandVectorData.O
MovingLegs.Move.OCommandVectorData.S OVirtualRobotComm.Effector.OCommandVectorData.O

#
# lostFoundSound -> ovirtualRobotAudioComm
#
LostFoundSound.Play.OSoundVectorData.S OVirtualRobotAudioComm.Speaker.OSoundVectorData.O
```

Each line of `connect.cfg` begins by the name a a subject, for example:

`NameOfObject.NameOfSubject.MessageType.S`

and ends by the corresponding observer, like:

`NameOfObject.NameOfObserver.MessageType.O`.

Note that the type of messages exchanged by the subject and the observer must be the same at each side.

2.3.3 Sending and receiving messages

In each C++ class of an OPEN-R object, subjects are referred by the `subject[]` array and observers by the `observer[]` array. To access this array convenient indexes are predefined by the *stubgen2* tool during the compiling process. The name of these indexes are the concatenation of the type of the service (`sbj` for subject and `obs` for observer) and of name of the service (the one declared in `stub.cfg`). For example `subject[sbjPlay]` refers to the Play gate of the LostFoundSound example (see section 2.3.1 .) `observer[obsCommand]` refers to the Command gate.

Sending a message

There are three steps to send a message:

1. initialize the message's content:
   `BallTrackingHeadResult result;`
   `result.status = BTH_SUCCESS;`
   Which can be rewritten shortly as:
   `BallTrackingHeadResult result(BTH_SUCCESS);`

2. assign the message to the service:
   `subject[sbjResult]->SetData(&result, sizeof(result));`

3. notify the observers:
   `subject[sbjResult]->NotifyObservers();`

Here is a sample code from the LostFoundSound example showing how to send a message:

```
...
/* Somewhere where the programmer wants to send a message */

BallTrackingHeadResult result(BTH_SUCCESS);
subject[sbjResult]->SetData(&result, sizeof(result));
subject[sbjResult]->NotifyObservers();

...
```

Receiving a message

In the service description done in the `stub.cfg`, the second function that appears at the end of the line is the function that is called by the operating system when a message arrives. To react when a message is received, this function must be implemented. For example, in the LostFoundSound example an observer is declared in the `stub.cfg` with the following line:
`Service:"LostFoundSound.Command.BallTrackingHeadCommand.O", null, NotifyCommand()`
The `NotifyCommand(const ONotifyEvent& event)` is the function called when a message arrives in the "Command" Gate of this object. This function proceeds with the following sequence:

1. retrieve the message's content by casting it:
   `BallTrackingHeadCommand* cmd = (BallTrackingHeadCommand*)event.Data(0);`

2. process the message.

3. send an AR message to the subject who sent the message:
   `observer[event.ObsIndex()]->AssertReady();`

Note that the object's observer index involved is conveyed by the message itself and the object uses this to make an explicit reference to this observer: `event.ObsIndex()`

The code is:

```
void LostFoundSound::NotifyCommand(const ONotifyEvent\& event)
{
   /* retrieving the message content */
   BallTrackingHeadCommand* cmd = (BallTrackingHeadCommand*)event.Data(0);

   /* use the cmd variable */

   /* send an AR message to the subject who sent the message */
   observer[event.ObsIndex()]->AssertReady();
}
```

Chapter 3

Getting information from the robot

The *OVirtualRobotComm* object has a subject that sends messages containing sensors information and a subject that sends images coming from the camera. We explain in this chapter how to retrieve this information.

3.1 Getting information from sensors

The outgoing gate (*aka* subject) of *OVirtualRobotComm* that sends informations from sensors is named `Sensor`. It sends message of the `OSensorFrameVectorData` type. The subject can be referred in `connect.cfg` using the following line:

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S.
```

3.1.1 The OSensorFrameVectorData data format

The OSensorFrameVectorData data format is a structure that contains information from all Aibo's sensors. The figure 3.1 shows a diagram of the OSensorFrameVectorData structure.

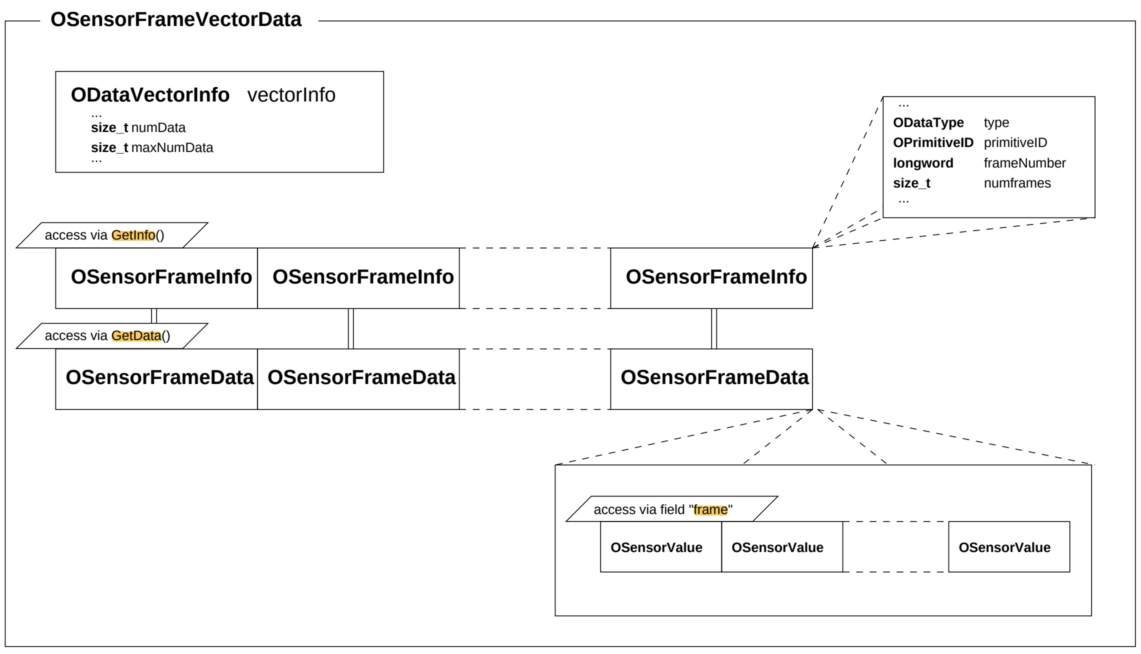

Figure 3.1: A diagram of the OSensorFrameVectorData data format.

The OSensorFrameVectorData structure contains three members: `vectorInfo` which is an ODataVectorInfo, an array of OSensorFrameInfo and an array of OSensorFrameData. The two arrays have an allocated memory which corresponds to a number of `vectorInfo.maxNumData` cells but their actual size is equal to `vectorInfo.numData` cells. Each cell of the OSensorFrameInfo array has a corresponding paired cell in the OSensorFrameData array (same index). These two cells contain the information and data for one given sensor during the last frames (a frame represents time in Aibo's hardware, see 2.1.2). The OSensorFrameData is a structure with a unique member which is an array of OSensorValue. The size of the OsensorValue array is the constant `osensorframeMAX_FRAMES`. Each cell of this array contains the value of the sensor for a frame, but the cells which contain valid data are the first `numframes` ones (`numframes` is a field of the OSensorFrameInfo paired with the current OSensorFrameData). The first cell of the OSensorValue array is tagged with the frame number `frameNumber` (field of the OSensorFrameInfo), the second one is tagged with `frameNumber+1`, etc. The OSensorFrameInfo structure has two members describing the type of the sensor and its ID (`primitiveID`. See next section), and two members describing the number of the first frame (`frameNumber`) and the number of valid frames (`numframes`).

Cells of the OSensorFrameInfo array are accessible using the `GetInfo(int index)` function, and those of the OSensorFrameData array are accessible using the `GetData(int index)` function. These two access functions are members of the OSensorFrameVectorData structure. `GetData()` has a return type of OSensorValue which is a generic data structure for sensor value. In fact the OSensorValue type is meant to be casted into one of the various types of sensor data that are described in the section 5.4.1, for example OJointValue for joints, OSwitchStatus for switches, etc.

3.1.2 Getting the right sensor

To access a sensor value we need to know the corresponding index of the OSensorFrameInfo and OSensorFrameData arrays. Let's first say few words about sensors in Aibo. Sensors and joints are also called *primitives* in the Sony's official documentation. In Aibo's design, each primitive can be referred 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.

Note that it is necessary to call the `OPENR::OpenPrimitive` function once in the program before accessing to a sensor, since it is also doing some initialization job.

Once we have the ID of the sensor, we have to iterate the OSensorFrameInfo array of the OSensorFrameVectorData structure and compare the member `primitiveID` of the cells with our ID to find the correct index. It is a good idea to store the index of sensors in a user array, so this is done only once.

To sum up, there are three steps to get the index of a sensor in the OSensorFrameInfo and OSensorFrameData arrays:

- Get the address of the sensor using the primitive locator.
- Get the primitive ID with OPENR::OpenPrimitive().
- Compare this primitive ID with the ones of the OSensorFrameInfo array to get the right index.
- Store this index in a user array.

Such an apparent complexity makes objects portable between different Aibo models (the same sensor can have a different index within two different models). That is why we do not recommend the use of the correspondance table between the index number and primitive locator provided in the Sony's model information document. Here is a sample code showing how to get the sensor index: In **MySampleClass.h**

```
...
// HEAD TILT JOINT SENSOR
static const char* const SENSOR_LOCATION = "PRM:/r1/c1-Joint2:j1"
...
class MySampleClass : public OObject {
    public:
...
    private:

    OPrimitiveID sensorID;
    int sensorIndex;
    bool sensorIndexInitialised;
...
```

In **MySampleClass.cc**

```
MySampleClass::MySampleClass()
{
   ...
   sensorIndex = -1;
   sensorIndexInitialised = false;
   ...
}

OStatus MySampleClass::DoInit( const OSystemEvent& event )
{
   ...

   /* At the stating of the object, call OpenPrimitive() */
   OpenPrimitive();
   ...
}


void MySampleClass::OpenPrimitive()
{
   /* Open the sensor primitive and get the sensor ID */
   OStatus result = OPENR::OpenPrimitive(SENSOR_LOCATION, &sensorID);
}

void MySampleClass::NotifySensor( const ONotifyEvent& event )
{
   OSensorFrameVectorData* sensorVec=(OSensorFrameVectorData*)event.Data(0);

   /* we need to know what is the sensor index in the OSensorFrameVectorData.
    * This is done once at the first time
    */
   if ( sensorIndexInitialised == false )
   {
      InitSensorIndex( sensorVec );
      sensorIndexInitialised = true;
   }

   ...
}

void MySampleClass::InitSensorIndex( OSensorFrameVectorData* sensorVec )
{
   /* iterate the vector to find the matching ID */
   for (int j = 0 ; j < sensorVec->vectorInfo.numData ; j++)
   {
      OSensorFrameInfo* info = sensorVec->GetInfo(j);
      if ( info->primitiveID == sensorID )
      {
         sensorIndex = j;
         break;
      }
   }
}
```

3.2 Getting information from the camera

Aibo supplies information from the camera through four different layers. Three layers are color images with a different resolution, the fourth is the color detection image (Aibo has an embedded parametrizable color detection algorithm, we will detail it later). Color 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 color detection image has only the color detection band which is a simple index. OPEN-R provides access to a pixel value in the Y band, the Cr band, the Cb band or the color detection band separately, but not to the pixel value coded as a whole in the YCrCb format. In other words, color images are treated as three monochromatic images. So to write a color image to the disk we have to retrieve the Y, Cr and Cb bands separatly and merge them.

The outgoing gate (aka subject) of *OVirtualRobotComm* that sends informations from the camera is named `FbkImageSensor`. It sends message of the `OFbkImageVectorData` type. The subject can be referred in `connect.cfg` using the following line:
`OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S`.

3.2.1 The OFbkImageVectorData data format

The OFbkImageVectorData data format is a structure containing several images. Figure 3.2 shows a diagram of this format. It contains three members: `vectorInfo` which is an ODataVectorInfo, an

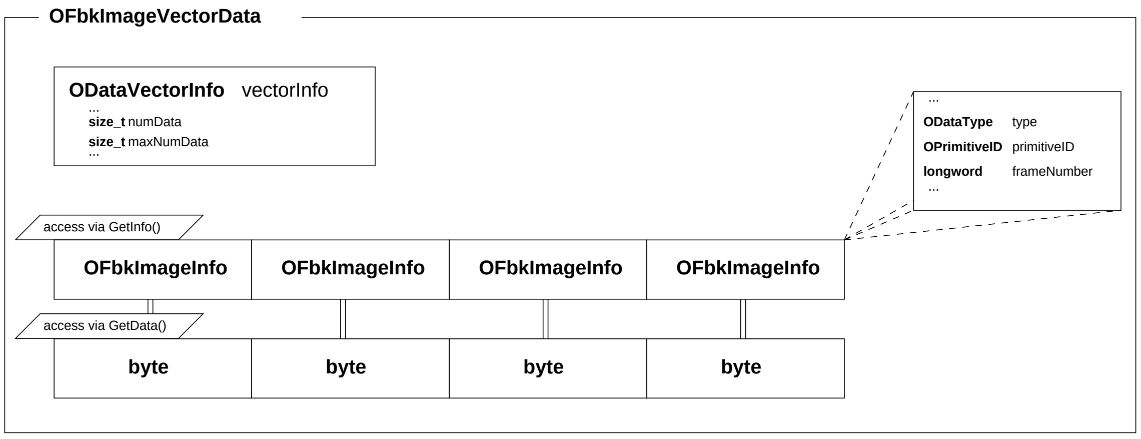

Figure 3.2: A diagram of the OFbkImageVectorData data format.

array of OFbkImageInfo and an array of bytes. The two arrays of the OFbkImageVectorData have a size of four, like the four layers we have presented in the previous paragraph. Each cell of the OFbkImageInfo array has a corresponding cell in the byte array. OFbkImageInfo contains information about the corresponding layer and the byte associated is a pointer to the image data for that layer. The index of the layer can be one of the following predefined constant: `ofbkimageLAYER_H` (color - high resolution), `ofbkimageLAYER_M` (color - medium resolution), `ofbkimageLAYER_L` (color - low resolution) and `ofbkimageLAYER_C` (color detection image).

Access to OFbkImageInfo cell is done via the `GetInfo()` function and access to the byte cell is done via the `GetData()` function. These function are members of the OFbkImageVectorData structure. For example to access to the medium resolution image layer use `GetInfo(ofbkimageLAYER_M)` and to access the layer data `GetData(ofbkimageLAYER_M)`. The next section explains how to access the different bands of the image once the image layer has been chosen.

3.2.2 Retrieving the Y, Cr, Cb or color detection band

Basic knowledge

The OPEN-R SDK provides a C++ class that handles image data: OFbkImage. Creating an instance of the OFbkImage class is necessary to get an access to one band of a layer. To create an OFbkImage, we need a pointer to the layer information and a pointer to the layer data. These pointers are retrieved using the `GetInfo()` and the `GetData()` functions of an OFbkImageVectorData, as we said previously. The constructor of the OFbkImage also needs an argument specifying the band (Y, Cr, Cb or color detection) that the OFbkImage will handle. For a color layer, to access the Y, Cr or Cb band, we will use respectively `ofbkimageBAND_Y`, `ofbkimageBAND_Cr`, `ofbkimageBAND_Cb`. For a color detection layer it will be `ofbkimageBAND_CDT`.

For example, if `fbkIVD` is of type OFbkImageVectorData, then `OFbkImage(fbkIVD->GetInfo(ofbkimageLAYER_M), fbkIVD->GetData(ofbkimageLAYER_M), ofbkimageBAND_Y)` will create an OFbkImage referring to the Y band of a medium resolution image layer.

`OFbkImage(fbkIVD->GetInfo(ofbkimageLAYER_C), fbkIVD->GetData(ofbkimageLAYER_C), ofbkimageBAND_CDT)` refers to the unique band of the color detection image layer.

Retrieving the three Y, Cr and Cb bands and merging them in a standard image format (like the BMP format) is needed to get a color image. The OPEN-R SDK does not provide any functions or class that handles standard image formats. However, there is BMP class available in the Sony's examples but we will not detail it here.

The OFbkImage class comes with some useful functions:

- `bool IsValid()`: returns `true` if OFbkImage is valid, else returns `false` if the pointers in arguments of the constructor are wrong.
- `byte* Pointer()`: returns a pointer to the beginning of an image data.
- `int Width()`: returns the width of an image.
- `int Height()`: returns the height of an image.
- `byte Pixel(int x, int y)`: returns the band value of the (x,y) pixel of an image. Which band depends on the parameter used in the OFbkImage constructor.
- `intSkip()`: returns the number of bytes to skip an entire line of an image.
- `byte ColorFrequency(OCdtChannel chan)`: returns the number of pixels (divided by 16) detected in the `chan` color detection band of a color detection image layer. (Color detection is explained next)

The following sample code shows the retrieving of a band in the function handling new image arrivals:

```
void Image::NotifyImage(const ONotifyEvent& event)
{
    /* first retrieve message information */
    OFbkImageVectorData* imageVec = (OFbkImageVectorData*)event.Data(0);

    /* then get the desired layer */
    OFbkImageInfo* info = imageVec->GetInfo(ofbkimageLAYER_C);
    byte* data = imageVec->GetData(ofbkimageLAYER_C);

    /* now get the desired band */
    OFbkImage cdtImage(info, data, ofbkimageBAND_CDT);

    /* do not forget to send back an AR */
    observer[event.ObsIndex()]->AssertReady();
}
```

Color detection

Aibo has a color detection algorithm built in. This algorithm is hardware encoded and is very fast. The color detection algorithm is explained in the section 4.3.2. For now, we just need to know that the color detection results are stored in the color detection layer.

Importantly, this fast color detection algorithm is programmable and can basically recognize up to 8 different colors. To do so, there are 8 different color detection channels programmable by the user: `ocdtCHANNEL0`, `ocdtCHANNEL1`, ..., `ocdtCHANNEL7`. The section 4.3.2 explains how to define a color. The color detection band contains information for the eight channels simultaneously, one per bit. A `&` operator with a bytemask can be used to check if a pixel is "on" in a color channel, that is if this pixel has been recognized as one of the corresponding color. The table 3.3 shows the different bytemasks for the color detection channels.

Channel number	bytemask	bytemask in hexadecimal
1	00000001	0x01
2	00000010	0x02
3	00000100	0x04
4	00001000	0x08
5	00010000	0x10
6	00100000	0x20
7	01000000	0x80
8	10000000	0x100

Figure 3.3: Table of color detection channel bytemasks.

For example, to know if the (10,10) pixel is "on" in the 5th color detection channel, use:

```
/*** fbkIVD is a OFbkImageVectorData ***/

// retrieve the color detection band of the color detection image layer.
OFbkImage channel(fbkIVD->GetInfo(ofbkimageLAYER_C),
                                    fbkIVD->GetData(ofbkimageLAYER_C),
                                    ofbkimageBAND_CDT);

// test is the (10,10) pixel is "on" or not in the channel 5
if ( (channel->Pixel(10,10) & 0x10) == 0x01 )
{
      // pixel is "on"
}
```

Chapter 4

Sending commands to the robot

In this chapter we will explain in detail how to send commands to Aibo. Commands can be sent to the robot to control the joints, the audio device and the camera.

4.1 Sending commands to joints

The incoming gate of *OVirtualRobotComm* that receives joints or LEDs commands is named `Effector`. It receives messages of the `OCommandVectorData` type. The observer can be referred in `connect.cfg` using the following line:

`OVirtualRobotComm.Effector.OCommandVectorData.O`.

4.1.1 The OCommandVectorData data format

The OCommandVectorData data format is a structure that contains commands for several Aibo's effectors (joints or LEDs). The figure 4.1 shows a diagram of the OCommandVectorData structure.

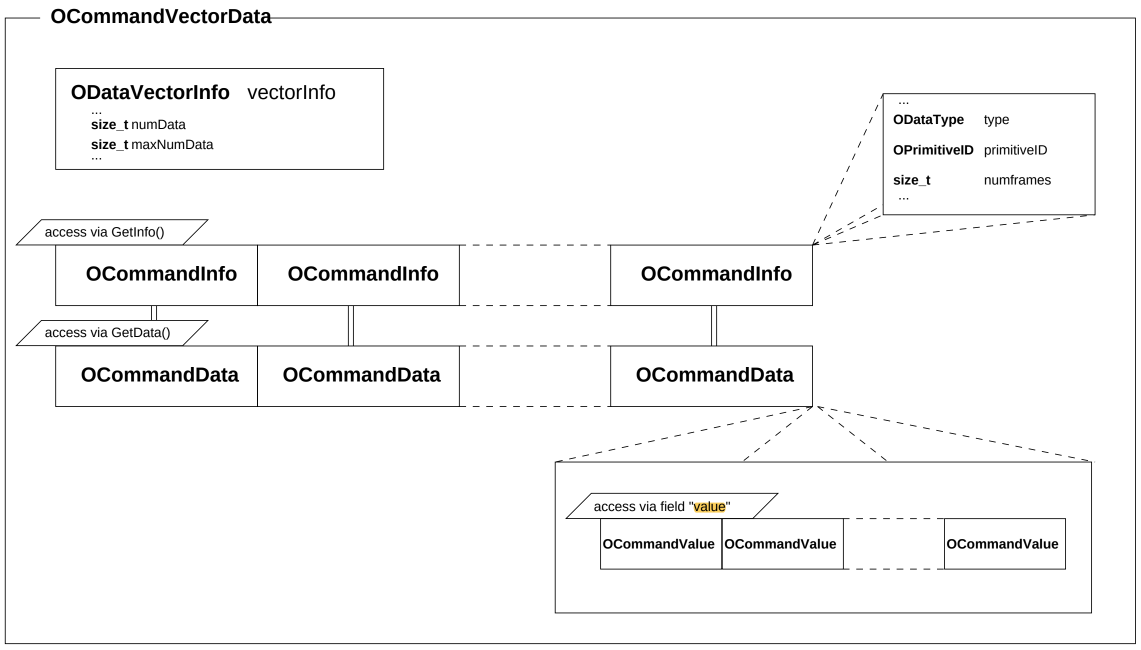

Figure 4.1: A diagram of the OCommandVectorData data format.

The OCommandVectorData structure contains three members: `vectorInfo` which is an ODataVectorInfo, an array of OCommandInfo and an array of OCommandData. These two arrays have an allocated memory size of `vectorInfo.maxNumData` cells. Each cell of the OCommandInfo array has a corresponding cell in the OCommandData array. These two cells contain the information and data for one effector command for the next frames (a frame represents time in Aibo's hardware, see 2.1.2). Since it is possible to send several frames for one effector, commands are somehow grouped. The OCommandInfo structure has two members describing the type of the effector and its primitiveID, and a member describing the number of frames passed in the command (`numframes`). The OCommandData is a structure with a unique member which is an array of OCommandValue. OCommandValue is a general type for sending a command for one frame to one effector. In fact OCommandValue must be cast to a type which is specific to the targeted effector. For example, OCommandValue2 is the type used to send a command to joints. The section 5.4.2 describes the different types used to send a command for a frame.

The size of the memory allocated for the OCommandValue array is `ocommandMAX_FRAMES` cells. As we said each cell of type OCommandValue contains the value of the effector for one frame. Only the first `numframes` (field of the corresponding OCommandInfo cell) cells will be read by the robot (the next ones will be ignored).

Cells of the OCommandInfo array are accessible using the `GetInfo(int index)` function, and those of the OCommandData array using the `GetData(int index)` function. These functions are members of the OCommandVectorData structure. The next section explains the common method used to send joint commands to OVirtualRobotComm.

4.1.2 Initialization steps

The BallTrackingHead example showed that an initialization sequence is needed before moving Aibo's joints. Three steps are needed for each joint that the program will use (joints can be initialized in any OPEN-R object):

1st step: getting the primitives ID

As sensors, each joint has an address in the primitive locator. This address must be converted to an OPrimitiveID in order to be exploitable. The `OPENR::OpenPrimitive()` does the conversion. The sample code below shows how to open two joints: In **MySampleClass.h**:

```
...
static const char* const JOINT_LOCATOR[] = {
    "PRM:/r1/c1-Joint2:j1",        // HEAD TILT
    "PRM:/r1/c1/c2-Joint2:j2"}     // HEAD PAN
...

class MySampleClass : public OObject{
...
private:
static const NUM_JOINTS = 2;

OPrimitiveID        jointID[NUM_JOINTS];
...
```

In **MySampleClass.cc**:

```
MySampleClass::OpenPrimitives()
{
    for (int i = 0; i < NUM_JOINTS; i++) {
        OStatus result = OPENR::OpenPrimitive(JOINT_LOCATOR[i],
                                              &jointID[i]);
        if (result != oSUCCESS)
        {
            /* error ! */
        }
    }
}
```

2nd step: setting joint gains

For each joint the motor gains (P, I and D gains) and shifts (P, I and D shifts) must be set up. In common programs these gains will be the standard one recommended by Sony (these values can be found in the Sony's model information document). Note that gain values are different for each joint. The programmer must call the `OPENR::EnableJointGain` function first so the gains become effective. Then, he can set them using the `OPENR::SetJointGain()` function which has the following prototype:
`OPENR::SetJointGain(OPrimitiveID primitiveID, word pg, word ig, word dg, word ps, word is, word ds).`

`pg, ig, dg` are respectively the P, I and D gains, and `ps, is, ds` are respectively the P, I and D shifts. Shifts are fixed values and should never be changed. The sample code below shows how to set the differents gains:
in **MySampleClass.h**

```
static const int    TILT_INDEX          = 0;
static const int    PAN_INDEX           = 1;

static const word   TILT_PGAIN          = 0x000a;
static const word   TILT_IGAIN          = 0x0008;
static const word   TILT_DGAIN          = 0x000c;

static const word   PAN_PGAIN           = 0x000d;
static const word   PAN_IGAIN           = 0x0008;
static const word   PAN_DGAIN           = 0x000b;

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

in **MySampleClass.cc**

```
void MySampleClass::SetJointGain()
{
    OPENR::EnableJointGain(jointID[TILT_INDEX]);
    OPENR::SetJointGain(jointID[TILT_INDEX],
                        TILT_PGAIN,
                        TILT_IGAIN,
                        TILT_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT);

    OPENR::EnableJointGain(jointID[PAN_INDEX]);
    OPENR::SetJointGain(jointID[PAN_INDEX],
                        PAN_PGAIN,
                        PAN_IGAIN,
                        PAN_DGAIN,
                        PSHIFT, ISHIFT, DSHIFT);
}
```

The P, I, D gains and shifts are hardware parameters for servo control. Since servo control knowledge is required to understand the consequences of changing gains and shifts, we recommend to use standard values. Failure to do so could result in damages for the robot.

3rd step: calibrating the joint

A gap between the real position and the position given by the sensor could exist when Aibo starts. The program must adjust this difference before moving joints. Reading a joint value with `OPENR::GetJointValue(OPrimitiveID primitiveID, *OJointValue value)` then setting the joint to this value by sending a command to the joint will correct the gap. Here is a sample code that adjusts joint gaps (`SetJointValue()` is a user-defined function that is explained in the example of the section 4.1.4.):

```
void MySampleClass::AdjustDiffJointValue()
{
    OJointValue current[NUM_JOINTS];

    for (int i = 0; i < NUM_JOINTS; i++)
    {
      OPENR::GetJointValue(jointID[i], &current[i]); // get the current
                                                        joint value
      SetJointValue(region[0], i,              // set the joint value
                    degrees(current[i].value/1000000.0),
                    degrees(current[i].value/1000000.0),
                    ocommandMAX_FRAMES);
    }

    /* Send the commands */
    subject[sbjMove]->SetData(region[0]);
    subject[sbjMove]->NotifyObservers();
}
```

4.1.3 Using the shared memory with the RCRegion class

In Sony's examples, a buffer method is used to send commands to OVirtualRobotComm. Why using a buffer? First a buffering method brings smoothness, but the main reason is that messages have a max size which is lower than the size of OCommandVectorData. Instead of sending a message of OCommandVectorData type, objects rather send a pointer to a cell of an OCommandVectorData array allocated in the shared memory. This pointer is of course much smaller and fits in the limits for an OCommandVectorData.

The OPEN-R SDK provides the RCRegion class that can access the shared memory segment and gives a reference counter. This counter holds the number of objects that have a pointer to a memory region allocated and behave as a mutex lock for this memory region.

In the particular case of sending commands to joints, it is the `OPENR::NewCommandVectorData()` function that allocates memory plus a RCRegion pointing to this memory, which is used to hold the reference counter. The `OPENR::NewCommandVectorData()` creates a OCommandVectorData and has three arguments:

- `size_t numCommands`: the number of cells in the OCommandData array.
- `MemoryRegionID* memID`: MemoryRegionID of the shared memory for OCommandVectorData.
- `OCommandVectorData** baseAddr`: pointer to OCommandVectorData.

This function initializes the value of the MemoryRegionID and the OCommandVectorData*. Once the `OPENR::NewCommandVectorData()` function has been called, the RCRegion class can be instanciated. The constructor that we will use is: `RCRegion(MemoryRegionID memID, size_t offset, void* baseAddr, size_t size)`. Several of the required arguments can be found in the ODataVectorInfo structure of the OCommandVectorData previously allocated.

- `memID` is the `memRegionID` member of the ODataVectorInfo.
- `offset` is the `offset` member of the ODataVectorInfo.
- `baseAddr` is the pointer initialized by OPENR::NewCommandVectorData().
- `size` is the `totalSize` member of the ODataVectorInfo.

The programmer must initialize the OCommandVectorData by setting the number of joints that will be controlled, the joints IDs, the number of frames for each joint. The sample code below shows the implementation for creating one OCommandVectorData for sending commands to 2 joints with the associated RCRegion class. In **MySampleClass.h**:

```
...
class MySampleClass : public OObject{
...
private:
    RCRegion* region;
...
```

In **MySampleClass.cc**:

```
void MySampleClass::NewCommandVectorData()
{
    OStatus result;
    MemoryRegionID      cmdVecDataID;
    OCommandVectorData* cmdVecData;
    OCommandInfo*       info;

    /* call NewCommandVectorData() */
    result = OPENR::NewCommandVectorData(NUM_JOINTS,
                                         &cmdVecDataID,
                                         &cmdVecData);

    if (result == oSUCCESS)
    {
      /* create a RCRegion pointing to the OCommandVectorData */
      region = new RCRegion(cmdVecData->vectorInfo.memRegionID,
                            cmdVecData->vectorInfo.offset,
                            (void*)cmdVecData,
                            cmdVecData->vectorInfo.totalSize);

      /* now initialize some of the OCommandVectorData members */
      cmdVecData->SetNumData(NUM_JOINTS);// number of OCommandData cells

      for (int j = 0; j < NUM_JOINTS; j++)  // for each OCommandInfo set:
      {
        info = cmdVecData->GetInfo(j);
        info->Set(odataJOINT_COMMAND2, // - the data type
                  jointID[j],          // - the ID of the joint to command
                  ocommandMAX_FRAMES); // - the number of frames
      }
    }
}
```

4.1.4 Setting a joint value

Once a OCommandVectorData and its corresponding RCRegion have been created it is possible to set joints values.

The first step is to be sure that no one else is reading the shared memory before writing it. The RCRegion class has a function `NumberOfReference()` that returns the number of references pointing to the allocated memory. If there is only one reference, it means that only the current object reads the allocated memory and so it is available for writing.

Then, as we already explained it, in the Sony Open-R SDK we must define a set of positions to reach for each frame and each joint included in the OCommandVectorData. The sample code below has a user function named `SetJointValue()`: it is an interface command which fills a joints values for a number of `ocommandMAX_FRAMES` frames in order to create a linear movement between a starting and an ending position. `rgn` is a pointer to a free RCRegion (`rgn->NumberOfReference() == 1`), `idx` is the index of the OCommandInfo and OCommandData arrays (thus `idx` points to a specific joint), `start` and `end` are the starting and ending values in degrees.

```
void
MySampleClass::SetJointValue(RCRegion* rgn, int idx, double start, double end)
{
    /* get a pointer to the OCommandVectorData from the RCRegion*/
    OCommandVectorData* cmdVecData = (OCommandVectorData*)rgn->Base();

    /* set members of the OCommandInfo cell */
    OCommandInfo* info = cmdVecData->GetInfo(idx);
    info->Set(odataJOINT_COMMAND2, jointID[idx], ocommandMAX_FRAMES);
    /* set frame values of the OCommandData cell */
    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); //value is converted in micro-radians
    }
}
```

Now we can send the command-message, for example:

```
/* assume that rgn points to a free region */
...
SetJointValue(rgn, TILT_INDEX, 0.0, 10.0);
SetJointValue(rgn, PAN_INDEX, 0.0, 10.0);

subject[sbjMove]->SetData(rgn);
subject[sbjMove]->NotifyObservers();
...
```

When the object *OVirtualRobotComm* receives the message, the reference counter of the RCRegion increase by one. The counter will decrease by one when *OVirtualRobotComm* processes the message and the subject can write again in the shared memory.

4.1.5 Controlling the pace of commands sending

There are two methods to control the pace of commands sending. The first is to send a message and wait for an AR message before sending another one. The second is faster and is useful in programs that need to be reactive (like tracking the pink ball with Aibo's head). This method uses a buffer that is an array of OCommandVectorData with the corresponding array of RCRegion. As the current move-command is processed by *OVirtualRobotComm*, the calling object tries to find a new free region in the array, set the joints' values for the next command in this region and send the message. This creates an autoregulated message queue: if *OVirtualRobotComm* speed to process commands is slower than the subject speed to send messages, soon none of the RCRegion will be free when the subject tries to find one and it will be stalled. It is only when a RCRegion is freed that the subject can send an other message. This ensures that no processing time will be lost in synchronization tasks since *OVirtualRobotComm* is constantly fed in.

4.2 Playing sound

The incoming gate of *OVirtualRobotAudioComm* that receives sound data is named `Speaker`. It receives message of the `OSoundVectorData` type. The observer can be referred in `connect.cfg` using the following line:
`OVirtualRobotAudioComm.Speaker.OSoundVectorData.O`.

4.2.1 The OSoundVectorData format

The OSoundVectorData data format is a structure containing sound information and data. Figure 4.2 shows a diagram of this format. It contains three members: `vectorInfo` which is an ODataVectorInfo, an array of OSoundInfo and an array of byte. The two arrays have a max size of `maxNumData`. Each cell of the OSoundInfo array has a corresponding cell in the byte array. These two cells contain the information and data for playing sound. The OSoundInfo structure has members describing the sound format and the primitive ID. The primitiveID is the speaker ID returned by `OPENR::OpenPrimitive()`. `frameNumber` is the frame number when the sound starts playing, `dataSize` is the size of the block containing the sound data. `format` is the sound format (it is always `osoundformatPCM`), `channel` is the number of channel in the sound data, `samplingRate`

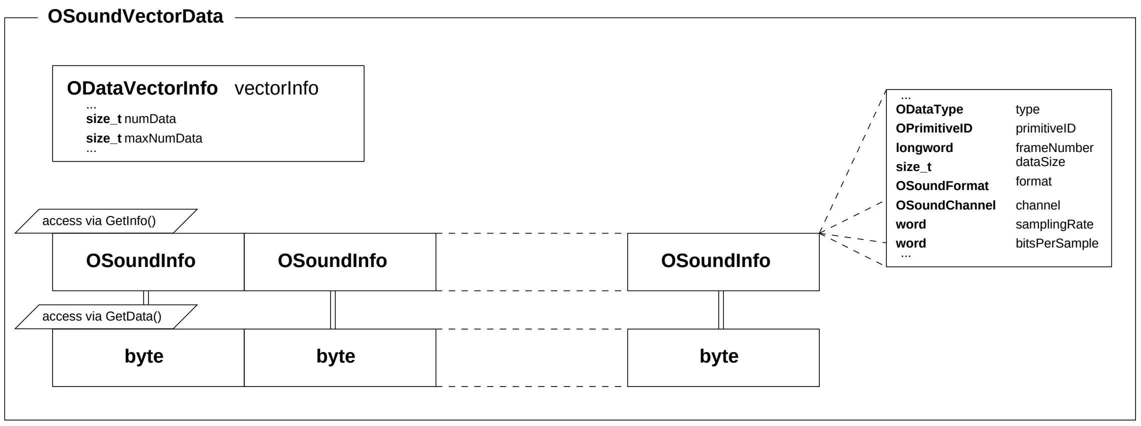

Figure 4.2: A diagram of the OSoundVectorData data format.

is the sampling rate and `bitsPerSample` is the number of bits per sample in the sound data. Each cell of the byte array is a pointer to the shared memory containing the sound data. The cells of the OSoundInfo array are accessible using the `GetInfo(int index)` function, and those of the OCommandData array are accessible using the `GetData(int index)` function. These functions are members of the OSoundVectorData structure. The next section explains the common method used to play sound.

4.2.2 Playing a .wav file

Since Aibo can only play sound in PCM format, a .wav file must be converted. In the samples provided by Sony there is a class called `WAV`, handling the 8KHz 8bits Mono wav format. This section describes the method for playing a .wav file but most of this method is common to all sound formats.

The sound is cut in blocks that are sent through a buffer to *OVirtualRobotAudioComm*. A buffering method is again necessary here because sound blocks can be larger than the max size of the messages. It also provides the ability to stop a sound currently playing: since a block currently playing cannot be stopped, we have to wait until it ends if we want to stop it. The smaller the block is, the lesser we will wait for its ending.

There are five steps required to play a sound:

1. Open the speaker to get its primitive ID
2. Create a OSoundVectorData buffer array in the shared memory.
3. Load the .wav file
4. Set the speaker volume
5. Send messages to *OVirtualRobotAudioComm* (play the sound)

The first four ones are initialization steps and are done only once.

Open the speaker

Like for any other primitives, the programmer needs to retrieve the speaker primitive ID before using it. To achieve this, use `OPENR::OpenPrimitive()`.

```
OPrimitiveID speakerID;
static const char* const SPEAKER_LOCATOR = "PRM:/r1/c1/c2/c3/s1-Speaker:S1";

void MySampleClass::OpenSpeaker()
{
    OStatus result = OPENR::OpenPrimitive(SPEAKER_LOCATOR, &speakerID);
}
```

Create a OSoundVectorData buffer

Since the sound data blocks are larger than what a message can hold, it is necessary to send the sound data in several steps. The way to do this is to create a OSoundVectorData array that will hold the whole sound data in several smaller blocks. This array buffering technique has already been presented in the previous section with motor commands. As we did in the section 4.1.3 we use a RCRegion based shared memory to hold the buffer.

First we have to create the buffer by setting several OSoundVectorData in the shared memory with the function `NewSoundVectorData()`. Its prototype is

`NewSoundVectorData(size_t numSounds, size_t dataSize, MemoryRegionID* memID, OSoundVectorData** baseAddr)`,

where `numSounds` is the size of the arrays in the OSoundVectorData structures to be created and `dataSize` is the size of the blocks containing sound data. This function sets `memID` and `baseAddr`. Once the array of OSoundVectorData is created we must create the corresponding array of RCRegion to hold them. We use the same RCRegion constructor as we did in section 4.1.3.

The following sample code shows how to create the sound buffer with blocks of 256 bytes length, calculated to last for 32ms:

```
// 8KHz 8bits MONO
// (8000 samples * 1 byte/sample * 1 channel * 32 ms = 256 bytes/ 32ms)
static const size_t SOUND_UNIT_SIZE  = 256;

static const size_t SOUND_NUM_BUFFER = 2;

RCRegion* region[SOUND_NUM_BUFFER];   // the sound buffer

void MySampleClass::CreateSoundBuffer()
{
    OStatus result;
    MemoryRegionID     soundVecDataID;
    OSoundVectorData* soundVecData;

    for (int i = 0; i < SOUND_NUM_BUFFER; i++) {

        result = OPENR::NewSoundVectorData(1, SOUND_UNIT_SIZE,
                                           &soundVecDataID, &soundVecData);
        // Set the size of OSoundInfo array to 1
        soundVecData->SetNumData(1);

        // Set OSoundInfo members
        // here speakerID is the one returned by OpenPrimitive()
        soundVecData->GetInfo(0)->Set(odataSOUND_VECTOR,
                                      speakerID, SOUND_UNIT_SIZE);

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

Load the .wav file

The .wav data files are located in the `OPENR/MW/DATA/P/` directory on the MemoryStick. Each file is associated to a keyword in the `OPENR/MW/CONF/DESIGN.DB` file. Here is an example of the `DESIGN.DB` file from the BallTrackingHead example:

```
FOUND_SOUND        /MS/OPEN-R/MW/DATA/P/FOUND.WAV
LOST_SOUND         /MS/OPEN-R/MW/DATA/P/LOST.WAV
```

In a program we can refer to a file using its associated keyword. The function
`OPENR::FindDesignData("KEYWORD", ODesignDataID* dataID, byte** addr, size_t* size)` loads the file corresponding to `"KEYWORD"` in a shared memory. This function also initializes `dataID` and allocates a shared memory of size `size` and sets the pointer `addr` to that memory. The ODesignDataID `dataID` holds information about the shared memory newly allocated by `OPENR::FindDesignData()`. This information is used when the file is unloaded from the shared memory with the `OPENR::DeleteDesignData(ODesignData DataID)` function.

An instance of a WAV class is linked to the wav file stored in the shared memory by using the `Set()` function of the WAV class. The sample code below shows how to load the previous `FOUND.WAV` file:

```
#include "WAV.h"

/* we must keep foundSoundID to free the shared memory
 * at the end of the program */
ODesignDataID  foundSoundID;
WAV foundWAV;

...
// Call the constructor of the WAV class
foundWAV()
...

void MySampleClass::LoadWAV()
{
    OStatus result;
    size_t size;
    byte*  addr;

    result = OPENR::FindDesignData("FOUND_SOUND", &foundSoundID, &addr, &size);

    foundWAV.Set(addr);

}
```

This global variable `foundWAV` must be defined since it will be later used by the `CopyWAVto` function.

Set the speaker volume

Before we play the sound we have to make sure that it will be audible. The OPEN-R API provides a static function `OPENR::ControlPrimitive()` that allows the programmer to control primitives like the camera, the speaker, etc. `OPENR::ControlPrimitive()` is often used to set parameters of the camera (see the following section) Here we will just show how to control the speaker volume. Only two of the `OPENR::ControlPrimitive()` parameters have to be set by the programmer to mute or unmute the speaker. The value of the first parameter is the ID of the speaker, the second is a constant that can be either `oprmreqSPEAKER_MUTE_ON` or `oprmreqSPEAKER_MUTE_OFF`. For example `OPENR::ControlPrimitive(speakerID, oprmreqSPEAKER_MUTE_ON, 0, 0, 0, 0);` mutes the speaker. When `oprmreqSPEAKER_MUTE_ON` is replaced by `oprmreqSPEAKER_MUTE_OFF` the speaker is enabled.

In order to set the volume we also use `OPENR::ControlPrimitive()` but more parameters need to be set. The first parameter is still the speaker ID, the second is the following constant:
`oprmreqSPEAKER_SET_VOLUME`.
The other parameters are a pointer to a OPrimitiveControl_SpeakerVolume (which is a special class that holds information about the speaker volume) and the size of this class. The OPrimitiveControl_SpeakerVolume class is instanciated with one of the following arguments: `ospkvolinfdB` (minimum), `ospkvol25dB` (-25dB), `ospkvol18dB` (-18dB), `ospkvol10dB` (-10dB, maximum.) The sample code below shows how to set the volume to -10dB:

```
OStatus result;
OPrimitiveControl_SpeakerVolume volume(ospkvol10dB);

result = OPENR::ControlPrimitive(speakerID,
                                    oprmreqSPEAKER_SET_VOLUME,
                                    &volume ,
                                    sizeof(volume) ,
                                    0 , 0);
```

Playing the sound

A buffering method is used to send sound data (OSoundVectorData) to *OVirtualRobotAudioComm*. Besides, the OSoundVectorData is not sent directly but is stored in a RCRegion based shared memory instead (as it was the case in the buffering method used in the joints command example). So, we first create a RCRegion based buffer. This buffer is of size 2, which is enough in that particular case, as we will explain it below.

The global variable `foundWAV` holds the whole wav sound data. It is capable of copying it by blocks of a specifyed size (256 octects) into a OSoundVectorData, or, more specificaly, into a RCRegion pointing to the OSoundVectorData. Doing so, it behaves as a sort of wav player, with a current index pointing to the ending of the last block sent. The function that performs this copying is `CopyWAVto(RCRegion *)`. The end of the wav file is reached when `CopyWAVTo()` returns `WAV_FAILS`, otherwise it returns `WAV_SUCCESS`. The first thing to do before using this `foundWAV` object is to "rewind" it so that the index points to the beginning of the wav sound data. The WAV class member `Rewind()` handles this job.

The buffering technique is used to insure a continuous flow of sound data towards *OVirtualRobotAudioComm*. We first fill the buffer (the two RCRegions) with the two first blocks of 256 octects (this is done in the `FillEntirelyBuffer` function in the example below, using the `CopyWAVto` function). Then, two messages are sent to *OVirtualRobotAudioComm*, with pointers to the two filled RCRegions. As soon as an AR message comes back from *OVirtualRobotAudioComm* (the `ReadPlay` function is started), we try to find one free RCRegion among the two of the buffer (`rgn[i]->NumberOfReference()==1 ?`), we fill it again with the next 256 octets of wav sound data (again, using `CopyWAVto`) and we send a new message pointing to this RCRegion.

This method ensures that the *OVirtualRobotAudioComm* will always have a block of sound to process.

The following code shows the entire playing process:

```
#include "WAV.h"

/* Fill entirely the sound buffer */
void MySampleClass::FillEntirelyBuffer
{
    for (int i = 0; i < SOUND_NUM_BUFFER; i++)
    {
        if (CopyWAVTo(region[i]) == WAV_SUCCESS)
            subject[sbjPlay]->SetData(region[i]);
    }
    subject[sbjPlay]->NotifyObservers();
}

/* this function is called when OVirtualRobotAudiocomm sends an AR */
void MySampleClass::ReadyPlay(const OReadyEvent& event)
{
    /* first find a free region */
    RCRegion* rgn = FindFreeRegion();

    /* try to copy a sound block in the buffer if
     * there are sound blocks remaining*/
    if (CopyWAVTo(rgn) == WAV_SUCCESS) {
        subject[sbjPlay]->SetData(rgn);
        subject[sbjPlay]->NotifyObservers();
    } else { // no blocks remaining
        if (IsAllRegionFree() == true) // the buffer is empty
        {
          /*  the sound has been played entirely */
        }
    }

WAVError MySampleClass::CopyWAVTo(RCRegion* rgn)
{
    OSoundVectorData* soundVecData = (OSoundVectorData*)rgn->Base();
    return foundWAV->CopyTo(soundVecData);
}

RCRegion* MySampleClass::FindFreeRegion()
{
    for (int i = 0; i < SOUND_NUM_BUFFER; i++) {
        if (region[i]->NumberOfReference() == 1) return region[i];
    }

    return 0;
}

bool MySampleClass::IsAllRegionFree()
{
    for (int i = 0; i < SOUND_NUM_BUFFER; i++) {
        if (region[i]->NumberOfReference() > 1) return false;
    }

    return true;
}
```

One difference between this buffering method and the one used in the joint commands that we explained before is that we use here AR messages from *OVirtualRobotAudioComm* to give rythm to the process. In the joint commands example, the rythm is given by incoming messages from the image gate of *OVirtualRobotComm* (FbkImageSensor gate).

4.3 Sending commands to the camera

As Aibo can move around in different environments, the camera shutter speed, gain or white balance has to be set accordingly to the lighting conditions.

The programmer can also define the color detection table used in the section 3.2.2. Here we will explain how to change the camera settings and color detection tables.

4.3.1 Setting the gain, color balance and shutter speed

The `OPENR::ControlPrimitive()` can be used to change the camera settings. We have to get the camera primitiveID to change settings (Use the `OPENR::OpenPrimitive()` function.)

To set the white balance, use:

```
OPrimitiveControl_CameraParam wb(ocamparamWB_OUTDOOR_MODE);
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_WHITE_BALANCE, &wb,
                        sizeof(wb), 0, 0);
```

The arguments of the OPrimitiveControl_CameraParam constructor can be :
`ocamparamWB_INDOOR_MODE`, `ocamparamWB_OUTDOOR_MODE`, `ocamparamWB_FL_MODE` (for fluorescent lamps)

To set the gain, use:

```
OPrimitiveControl_CameraParam gain(ocamparamGAIN_MID);
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_GAIN, &gain, sizeof(gain),
                        0, 0);
```

arguments of the OPrimitiveControl_CameraParam constructor can be: `ocamparamGAIN_LOW`, `ocamparamGAIN_MID`, `ocamparamGAIN_HIGH`.

To set the shutter speed, use:

```
OPrimitiveControl_CameraParam shutter(ocamparamSHUTTER_FAST);
OPENR::ControlPrimitive(cameraID, oprmreqCAM_SET_SHUTTER_SPEED, &shutter,
                        sizeof(shutter), 0, 0);
```

arguments of the OPrimitiveControl_CameraParam constructor can be: `ocamparamSHUTTER_SLOW`, `ocamparamSHUTTER_MID`, `ocamparamSHUTTER_FAST`

Since the camera sensitivity is not very high, we recommand to use slow shutter speed and high camera gain in most indoor environments.

4.3.2 Setting a color detection table

The programmer can set up to eight color detection channels that can be used by the internal color detection algorithm as we explained before (see 3.2.2). As we said in the section 3.2 the colors in Aibo are in the YCrCb format. A color description for a given channel is built on a set of 32 CrCb plans spreading along the Y component of the color space. For each of the 32 values along Y, a rectangle is defined in the corresponding CrCb plan. During the color detection process, Aibo's harware takes the Y component at each pixel and test if the corresponding (Cr,Cb) value falls within the rectangle which is defined for this value of Y in the color description of each channel of the detection table and thus detect which channel this pixel belongs to (it can be multiple channels). Figure 4.3 shows a 3D representation of this method of color description in the (Y,Cr,Cb) color space.

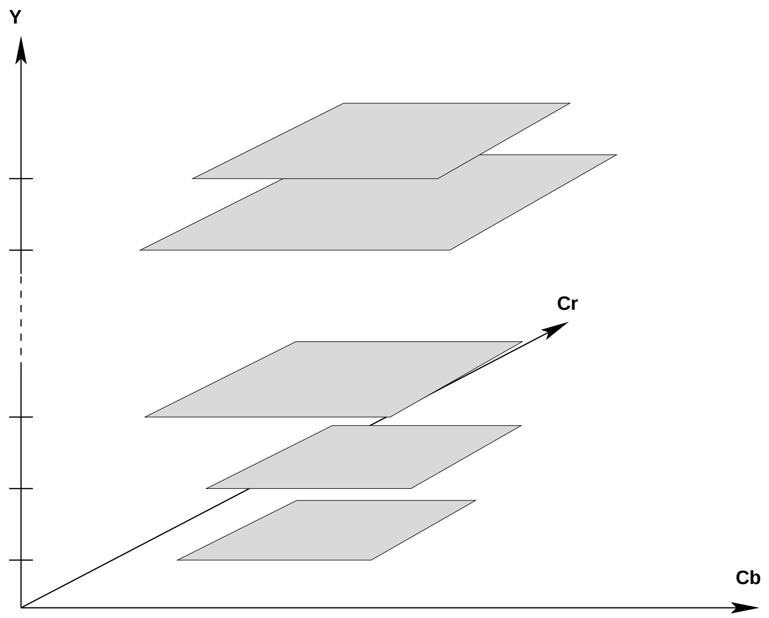

Figure 4.3: 3D representation of color matching rectangles in the (Y,Cr,Cb) reference .

A special method is used to set a color table and communicate it to FbkImageSensor. First we have to create a OCdtVectorData in a shared memory with `OPENR::NewCdtVectorData()`. OCdtVectorData is a data structure that holds a color detection table or "Cdt" (see figure 4.4).

It has two members: an ODataVectorInfo and an array of eight OCdtInfo. An OCdtInfo holds the information for one channel of the color detection table. Access to an OCdtInfo is granted via the `GetInfo()` function of the OCdtVectorData structure.

Each of the eight color detection channels is represented by an OCdtInfo. We must init the OCdtInfo with the `Init()` function before using it. Then we can set information with the `Set()` command. This command is straightforward, as it gives first the value of Y and then the bounding rectangle for the CrCB plan.

Once the color detection table has been set for the desired channels, we call the `OPENR::SetCdtVectorData()` function that “sends” the OCdtVectorData() to the OfbkImageSensor. To finish, we release the shared memory with `OPENR::DeleteCdtVectorData()`. The sample code below shows the entire method to set one channel in the color detection table.

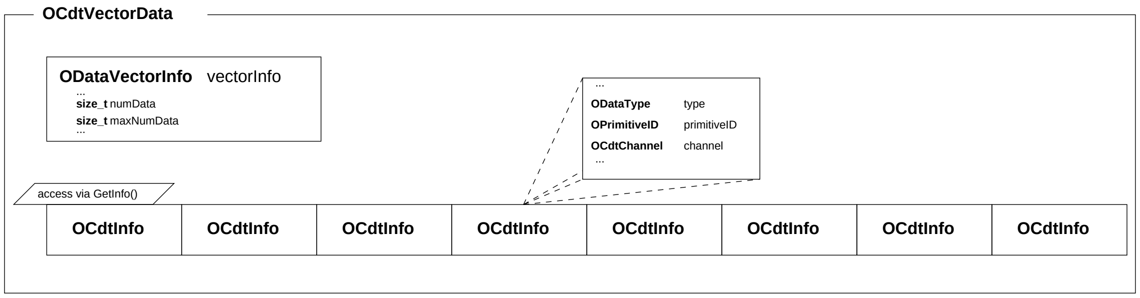

Figure 4.4: A diagram of the OCdtVectorData data format.

```
#define CHANNEL_0 0

/* fbkID is the primitive ID of the fbkImageSensor */

void MySampleClass::SetCdtVectorDataOfPinkBall()
{

    OStatus result;
    MemoryRegionID   cdtVecID;
    OCdtVectorData*  cdtVec;
    OCdtInfo*        cdt;

    // create the OCdtVectorData in the share memory
    result = OPENR::NewCdtVectorData(&cdtVecID, &cdtVec);
    if (result != oSUCCESS) {
        // error !
        return;
    }

    // indicate that we set up 1 channel
    cdtVec->SetNumData(1);

    // retrieve the first cell of the OCdtInfo array
    cdt = cdtVec->GetInfo(0);

    /*initialize the OCdtInfo. Here we will use the
     * channel 0 */
    cdt->Init(fbkID, CHANNEL_0);

    /* Set the color table with each Y segment.
     * cdt->Set(Y_segment, Cr_max,  Cr_min, Cb_max, Cb_min)
     */
    cdt->Set( 0, 230, 150, 190, 120);
    cdt->Set( 1, 230, 150, 190, 120);
    cdt->Set( 2, 230, 150, 190, 120);
    cdt->Set( 3, 230, 150, 190, 120);
    cdt->Set( 4, 230, 150, 190, 120);
    cdt->Set( 5, 230, 150, 190, 120);
    cdt->Set( 6, 230, 150, 190, 120);
    cdt->Set( 7, 230, 150, 190, 120);
    cdt->Set( 8, 230, 150, 190, 120);
    cdt->Set( 9, 230, 150, 190, 120);
    cdt->Set(10, 230, 150, 190, 120);
    cdt->Set(11, 230, 150, 190, 120);
    cdt->Set(12, 230, 150, 190, 120);
    cdt->Set(13, 230, 150, 190, 120);
    cdt->Set(14, 230, 150, 190, 120);
    cdt->Set(15, 230, 150, 190, 120);
    cdt->Set(16, 230, 150, 190, 120);
    cdt->Set(17, 230, 150, 190, 120);
    cdt->Set(18, 230, 150, 190, 120);
    cdt->Set(19, 230, 150, 190, 120);
    cdt->Set(20, 230, 160, 190, 120);
    cdt->Set(21, 230, 160, 190, 120);
    cdt->Set(22, 230, 160, 190, 120);
    cdt->Set(23, 230, 160, 190, 120);
    cdt->Set(24, 230, 160, 190, 120);
    cdt->Set(25, 230, 160, 190, 120);
    cdt->Set(26, 230, 160, 190, 120);
    cdt->Set(27, 230, 160, 190, 120);
    cdt->Set(28, 230, 160, 190, 120);
    cdt->Set(29, 230, 160, 190, 120);
    cdt->Set(30, 230, 160, 190, 120);
    cdt->Set(31, 230, 160, 190, 120);

    // send the command
    result = OPENR::SetCdtVectorData(cdtVecID);
    if (result != oSUCCESS) {
        // error !
    }

    // release the memory
    result = OPENR::DeleteCdtVectorData(cdtVecID);
    if (result != oSUCCESS) {
        // error !
    }
}
```

Chapter 5

Miscellaneous

5.1 Printing to the wireless console

Aibo can be connected in a wireless environnement. In fact, it is highly recommanded since it facilitates the debuging and monitor processes. For this purpose, a remote console is available. The wireless console is available when the Wconsole environment configuration has been choosen for the Memory Stick (see 5.2.2). Then, we can easily access this console using the telnet application on the port 59000:

`telnet Aibo_IP 59000`.

Within OPEN-R programs, it is possible to print in the wireless console. There are two macros defined in the OPEN-R SDK that simplify this printing operation. The first is `OSYSPRINT()`, and is always available while the other, `OSYSDEBUG()`, is only available when the `OPENR_DEBUG` compilation flag is defined when compiling with *gcc*. The use of `OSYSPRINT()` and `OSYSDEBUG()` is similar to the `printf()` function in the standard C library. Here are some examples:

```
OSYSPRINT(("Hello world !\n"));
// prints Hello world ! and begins a new line

OSYSDEBUG(("The joint's angle is: %d degrees\n", jAngle));
//  prints The joint's angle is: 2.2564 degrees, and begins a new line.
```

5.2 How to compile and execute an OPEN-R program

In this section we will not explain how to install the OPEN-R SDK but we will only describe how to compile and run an OPEN-R program. Information about the installation of the OPEN-R SDK can be found at the OPEN-R official web site[1].

5.2.1 Compilation

Before calling *gcc* in the makefile, we must set up some files and run some OPEN-R SDK tools. These tools generate intermediate files that are required to compile an OPEN-R object C++ class. We will describe the compilation procedure for one OPEN-R object.

files to be set

The first file we have to set is the `stub.cfg` of the object as described in the section 2.3.1. This file must be in the same directory than the .cc and .h of the object. Then we have to set the `objectName.ocf` file. This file contains useful information used when linking the librairies. The file contains one line with the following format: `object` *objectname stack_size heap_size sched_priority cache tlb mode*

[1]http://openr.aibo.com in the member area.

- *objectname*: the name of the object.
- *stack_size*: the size of the stack, in bytes.
- *heap_size*: the size of which the heap will be extended if the object runs out of heap space.
- *sched_priority*: the scheduling priority of the object. Normally leave this value to "128".
- *cache*: the state of using the processor's memory cache. It is either "cache" or "nocache". "cache" is recommended.
- *tlb*: the allocation area of the object's memory. if "tlb" is specified, the memory will be allocated in the virtual address space. With "notlb" the memory is allocated in the physical address space.
- *mode*: the execution mode of the object. It is either "user" for user-mode or "kernel" for kernel mode.

Usually objects can be linked with standard settings. The configuration line in `objectName.ocf` with standard settings is:
`object objectName 3072 16384 128 cache tlb user`

compiling

Once `stub.cfg` and `objectName.ocf` have been set up, we can launch the compilation. We advise you to create a makefile for each object. The following code is an example of makefile that the programmer can easily modify to his own taste by changing "mySampleClass" to his/her object's name:

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

#
# When OPENR_DEBUG is defined, OSYSDEBUG() is available.
#
# CXXFLAGS+= -DOPENR_DEBUG

.PHONY: all install clean

all: mySampleClass.bin

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

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

mySampleClass.bin: MySampleClassStub.o .o mySampleClass.ocf
        $(MKBIN) $(MKBINFLAGS) -o $@ $^ $(LIBS)
        $(STRIP) $@

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

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

When we type `$> make`, first `stungen2` is called, then `mipsel-linux-g++` compiles. The linking is done by the OPEN-R `mkbin`. When typing `$> make install`, the bin object is compressed with `gzip` and then moved to the `$(INSTALLDIR)/OPEN-R/MW/OBJS/` directory.

5.2.2 Execution on Aibo

There are two steps to perform before running an OPEN-R program on Aibo. First, the base system must be installed on the memory-stick, and then, on top of this base system, we can copy our own OPEN-R objects and set up config files that will launch our OPEN-R program when Aibo boots.

preparing the Memory Stick

We have to copy a base OPEN-R system able to run OPEN-R programs on the Aibo's Memory Stick. Three configurations are avaible:

- Basic: without a wireless LAN environment
- Wlan: with a wireless environment but without a console.
- Wconsole: with a wireless environment and console.

On top of that we can choose between memory protection (memprot) or not (nomemprot). By default we suggest to choose memory protection because it is safer. Once we have choosen the configuration, we have to copy to the root of the Memory Stick the base system which is available in the corresponding `OPEN-R` directory. Directories that can be copied are situated in `/usr/local/OPEN_R_SDK/OPEN_R/MS/`:

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

Copying objects to the Memory Stick and setting config files

For each objects we have to copy the corresponding `.bin` files from the
`$(INSTALLDIR)/OPEN-R/MW/OBJS/` directory to the `/OPEN-R/MW/OBJS/` directory on the Memory Stick. At this time the `/OPEN-R/MW/CONF/OBJECT`.
`CFG` file on the Memory Stick must be edited: it contains the name of the objects that will be executed. For example:

```
/OPEN-R/MW/OBJS/MYSMPLE.bin
/OPEN-R/MW/OBJS/TOTO.bin
```

Usually, this is done once when the objects are first created and then, it is enough just to copy the .bin files to update the program.

We also have to put the `connect.cfg` and `designdb.cfg` files in the `/OPEN-R/MW/CONF` directory on the Memory Stick. If we choosed to use a wireless environment, we would have to set the `WLANCONF.txt` file. Here is an sample configuration of `WLANCONF.txt`:

```
HOSTNAME=AIBO1
ETHER_IP=192.168.1.10
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.1.1
ESSID=linksys
WEPENABLE=0
WEPKEY=AIBO2
APMODE=1
CHANNEL=6
DNS_SERVER_1=147.250.1.1
DNS_DEFDNAME=ensta.fr
```

Running the program on Aibo

Simply put the Memory Stick in Aibo and boot it up.

5.3 Using the FTP protocol to transfer file into Aibo

Extracting and reinserting the Memory Stick in the robot can become repetitive when the programmer debugs an OPEN-R program. However, there is an OPEN-R object called *TinyFTPD* that provides a FTP server on Aibo. This object in part of the sample programs provided by Sony. After compiling it (`make`, `make install` in the `samples/TinyFTPD/` directory), put the binary file `samples/TinyFTPD/MS/OPEN-R/MW/OBJS/TINYFTPD.BIN` in the `/OPEN-R/MW/OBJS/` directory on the Memory stick and add the line `/OPEN-R/MW/OBJS/TINYFTPD.bin` in the `OBJECT.CFG` file situated in the `/OPEN-R/MW/CONF/` directory (see previous section).

We also have to put the `samples/TinyFTPD/MS/OPEN-R/MW/CONF/PASSWD` file in the `/OPEN-R/MW/CONF/` directory on the memory stick.

Then, when Aibo is running, execute a FTP client: `ftp AIBO_IP`. You can log in using the anonymous account (login: `anonymous`, password: `anonymous` .) The usual FTP commands are available (`PUT`, `GET`, etc.) In order to reboot Aibo remotely, use the following command: `QUOTE REBT` (***!!! Beware because Aibo can fall as it reboots !!!***).

5.4 data structures

In this section we will describe the data structures that are specific to each type of sensor (joints, switches, etc.) and each type of command (joints, LEDs, etc.)

5.4.1 Sensors

The general type OSensorValue has to be cast to the sensor's specific type for retrieving information from this sensor (see 3.1.1).

OAcceleration

OAcceleration is a data type that holds acceleration values from the x, y or z acceleration sensor. The useful member is `value`. The unit is $10^{-6} m.s^{-2}$. The member `signal` is the A/D signal from the sensor.

```
struct OAcceleration
{
    slongword  value;
    word       signal;
    word       padding[5];
}
```

OTemperature

OTemperature is a data type that holds the temperature value returned by the temperature sensor. The useful member is `value`. The unit is $10^{-6o}C$. The member `signal` is the A/D signal from the sensor.

```
struct OTemperature
{
    slongword   value;
    word        signal;
    word        padding[5];
}
```

OPressure

OPressure is a data type that holds the pressure value returned by the pressure sensor. The useful member is `value`. The unit is in $10^{-6}N.m^{-2}$. The member `signal` is the A/D signal from the sensor.

```
struct OPressure
{
    slongword   value;
    word        signal;
    word        padding[5];
}
```

OLength

OLength is a data type that holds the distance value returned by the ultrasonic distance sensor. The useful member is `value`. The unit is $10^{-6}m$. The member `signal` is the A/D signal from the sensor.

```
struct OLength
{
    slongword   value;
    word        signal;
    word        padding[5];
}
```

OSwitchStatus

OSwitchStatus is a data type that holds the status returned by a switch. The useful member is `value`. It is either oswitchON or oswitchOFF. The member `signal` is the A/D signal from the sensor.

```
struct OSwitchStatus
{
    OSwitchValue   value;
    word           signal;
    word           padding[5];
}
```

OJointValue

OJointValue is a data type that holds the angle value returned by a joint. The useful member is `value`. The unit is $10^{-6} rad$. The member `signal` is the feedback signal from the sensor (which is converted using a table to obtain `value`). The other members are unuseful for standard programs.

```
struct OJointValue
{
    slongword   value;
    word        signal;
    sword       pwmDuty;
    slongword   refValue;
    word        refSignal;
    word        padding[1]
}
```

5.4.2 effectors

The general type OCommandValue has to be cast to the effector's specific type for sending a command to this sensor (see 4.1.1).

OJointCommandValue2

OJointCommandValue2 is a data type that holds the command for a joint for 1 frame. The useful member is `value`. The unit is $10^{-6} rad$.

```
struct OJointCommandValue2
{
    slongword   value;
    slongword   padding;
}
```

OJointCommandValue3

OJointCommandValue3 is a data type that holds the command for the plunger movement in the ears of an ERS-210 for 1 frame. The useful member is `value`.It can be `ojoint3_STATE0` or `ojoint3_STATE1`.

```
struct OJointCommandValue3
{
    OJointValue3   value;
    word           reserved;
    word           padding;
}
```

OLEDCommandValue2

OLEDCommandValue2 is a data type that holds the command for a LED for 1 frame. The useful members are `value` (ON / OFF, respectively `oledON` and `oledOFF`) and `period` (how long a LED will remain in the state. The unit is 8ms).

```
struct OLedCommandValue2
{
    OLedValue   led;
    word        period;
    word        reserved;
}
```

Bibliography

[1] Sony. *OPEN-R SDK Documents*. http://openr.aibo.com/ (download section of the registered area.)

[2] Sony. *OPEN-R SDK school* (6 tutorials). http://openr.aibo.com/ (OPEN-R SDK University section of the registered area.)

[3] AiboHack web site: http://www.aibohack.com