Remote control of the Aibo^{TM} camera from Webots^{TM}

Raphaël Haberer-Proust

February 20, 2006

Semester project, winter 2005-2006

School of Computer & Communication Sciences
Swiss Federal Institute of Technology Lausanne (EPFL)

Supervisor: Olivier Michel, Cyberbotics Ltd.
Responsible Professor: Auke Jan Ijspeert, BIRG EPFL

Abstract

This is the report of a semester project done with the Biologically Inspired Robotics Group (BIRG) at the Swiss Federal Institute of Technology Lausanne (EPFL) during the winter semester 2005/2006. The goal of this project was to extend existing software in order to be able to access the images of the Aibo robot's camera.

Contents

List of Figures

List of Tables

Listings

Acknowledgment

First of all, I would like to thank Olivier Michel for always have been available, his transmission of technical skills and for showing me the way to go. Then, I would also like to thank Ricardo A. Téllez for his encouragement and his immediate readiness to share his knowledge of OPEN-R. Special thanks go to Alessandro Crespi and Fabrice Haberer-Proust for their help in programming. Last but not least, I would like to express my gratitude to Professor Auke Jan Ijspeert for letting me work for the Biologically Inspired Robotics Group (BIRG) and for providing me an introduction to all the interesting projects and kind people there.

About trademarks

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

Chapter 1

Introduction

1.1 Aibo

Aibo is an interactive quadruped dog-shaped robot produced by Sony. Despite the fact that it was principally designed as an entertainement system, it is widely used in research fields because of its great features. Indeed, it is provided with 18 individually controllable joints, distance sensors, touch sensors, LEDs and even a colour camera. The most interesting point is that it is programmable and able to communicate over a standard wireless LAN card [14]. Thanks to its 64bit MIPS processor running at a clock speed of 576MHz, Aibo has enough computational power to allow the programmer for example, to make it move or walk while it is executing some other programmes, like voice recognition or evolutional algorithms.

Figure 1.1: Aibo ERS-7 robot

For software development, Sony provides the OPEN-R SDK and a very com-

plete documentation [8] including sample programmes [10].

1.2 Webots

Webots is a three-dimensional mobile robot simulation produced by Cyberbotics Ltd. and co-developed by the Swiss Federal Institute of Technology Lausanne (EPFL) [1] [2]. It is available for Linux i386, Mac OS X and Windows. It allows the user to:

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

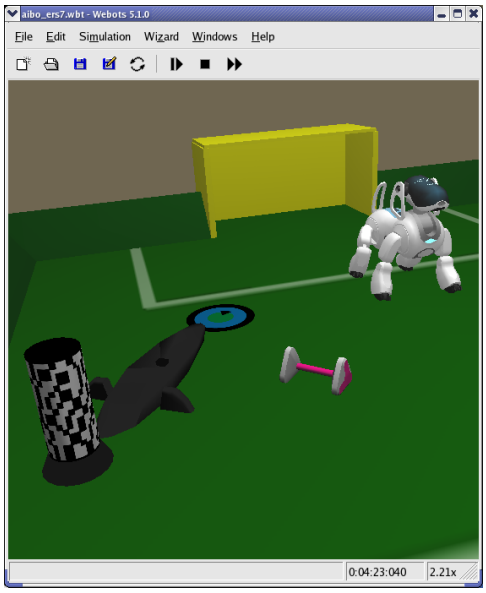

Figure 1.2: Webots with the `aibo_ers7.wbt` world

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

Furthermore, the simulation model includes a tool that allows one to remotely control the Aibo, both the simulated model and the real Aibo either separately or simultaneously. This tool is simply called *remote control*. When using it with a real Aibo, the communication is achieved through its wireless LAN interface. When using it with the simulated version, the commands are directly sent through software.
The remote control is accessible by double-clicking on the Aibo model in the main Webots window showing the 3D world. Its main window then pops up containing the buttons and fields for configuring and setting up the communication with the real Aibo. For the control of the simulated model, this tab has no use. Under the tab "Control" one can find slide controllers that let one control all the 18 servo motors of Aibo. Each servo of both the simulated and the real Aibo can be controlled either individually or several simultaneously.

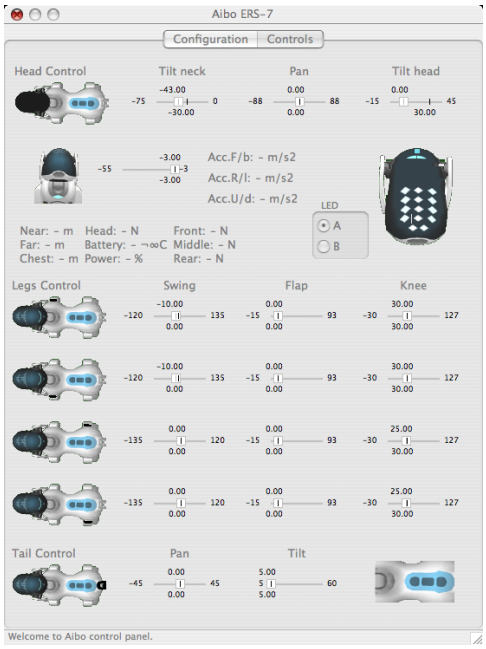

Figure 1.3: Webots' Aibo ERS-7 remote control, control tab

1.3 RCServer

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

on the network, a connection can be established in order to remotly control the real Aibo robot.

1.4 Project objectives

The goal of this project was to integrate camera support into Webots' remote control for Aibo. This means that the user should be able to either take a picture of what Aibo's camera sees when the user presses the "take a picture" button in the remote control window, or to view in real time a video stream that Aibo's camera captures.

Chapter 2

System Overview

This section describes the elements of the system that are important to know before trying to understand how the camera support was implemented. It does not treat how to set up the programming environment, or how to run a program on Aibo. These information can be found under [11] and [16]. Details about the implementation itself are discussed in the next chapter, chapter 3.

2.1 Aibo

For the development and the tests, the lab's Aibo ERS-7 (Figure 1.1) was available. However, the RCServer code was designed to work on other Aibo models like the ERS-210. The functionality of the camera support should also work with few or even no modifications on other Aibo models, but this was not tested since no other models were available.

2.1.1 OPEN-R

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

Features of OPEN-R:

- *Modularized software and inter-object communication*
  OPEN-R software is object-oriented and modular. Software modules are called "objects" (specifically, "OPEN-R objects").
  Processing is performed by multiple objects with various functionalities running concurrently and communicating via inter-object communication.
  Connections between objects are defined in an external description file

(called `stub.cfg`, see the paragraph about it later). When the system software boots, the description file is loaded and used to allocate and configure the communication paths for inter-object communication. Connection ports in objects are identified by the service name, which enables objects to be highly modular and easily replaceable as software components.

- *Layered structure of the software and services provided by the system layer*
  The OPEN-R system layer provides a set of services (input of sound data, output of sound data, input of image data, output of control data to joints, and input of data from various sensors) as the interface to the application layer. This interface is also implemented by inter-object communication.
  OPEN-R services enable application objects to use the robot's underlying functionalities, without requiring detailed knowledge of the robot hardware.
  The system layer also provides the interface to the TCP/IP protocol stack, which enables programmers to create networking applications utilizing the wireless LAN. The IPStack is an OPEN-R system layer object. Objects can use the network services offered by the IPv4 protocol stack by communicating with the protocol stack through normal message passing, i.e. by sending special messages to and receiving special messages from the IPStack.

Object

OPEN-R application software consists of several OPEN-R objects. These are not objects in the object-oriented sense of the word. The concept of an object is similar to one of a process in the UNIX or Windows operating systems with regard to the following points of view. Characteristics specific to objects:

- *An object corresponds to one executable file.*
  An object is a concept that only exists at run-time. Each object has a counterpart in the form of an executable file, created at compile-time. Source code is compiled and linked to create this executable file. Then, the file is put on an Aibo Programming Memory Stick. When Aibo boots, the system software loads the file from the Memory Stick and executes it as an object. An executable file usually has a filename with a `.bin` extension.

- *Each object runs concurrently with other objects.*
  Each object has its own thread of execution and runs concurrently with other objects in the system.

- *Objects exchange information using message passing.*
  An object can send messages to other objects. When an object receives a message, the method corresponding to the message is invoked, with the data in the message as its argument.
  An important feature of objects is that they are single-threaded. This means that an object can process only one message at a time. If an object receives a message while it is processing another message, the second message is put into the message queue and processed later. The typical life cycle of an object:

  1. Loaded by the system
  2. Waits for a message
  3. When a message arrives, executes the corresponding method. Possibly sends some messages to other objects.
  4. When the method finishes execution, goes to step 2.

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

- *An object has multiple entry points.*
  Unlike an ordinary programming environment in which a program has a single entry point `main()`, OPEN-R allows an object to have multiple entry points. Each entry point corresponds to a method. Some entry points are common to all objects and have purposes that are determined by the system, e.g. initialization and termination. Other entry points are specific to a certain object.

Inter-object communication

The use of inter-object communication enables each object to be created separately and later be connected to other objects. When two objects communicate, the side that sends data is called the "subject" and the side that receives data is called the "observer". The subject sends a 'NotifyEvent' to the observer. NotifyEvent includes the data that the subject wants to send to the observer. The observer sends a "ReadyEvent" to the subject. The purpose of ReadyEvent is to inform the subject whether the observer is ready to receive data or not (see Figure 2.1). If the observer is not ready to receive data, the subject should not send any data to the observer, otherwise messages might be ignored[15].
In the OPEN-R SDK terminology, subjects and observers are called *services*.

The virtual objects `OVirtualRobotComm` and `OVirtualAudioComm`

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

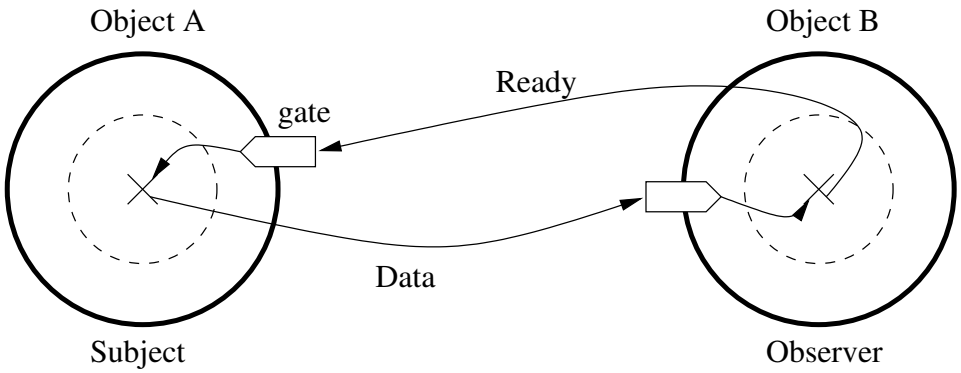

Figure 2.1: Inter-object communication

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

`OVirtualRobotAudioComm` interfaces with the robot audio devices

The use of those objects in the programmes is the same as the use of programmer defined objects. The gates in those objects are already defined to send messages to and receive messages from them [16].

Description of the file `stub.cfg`

The connection between the entry points (where the message is received) and the actual member functions of the core class is described with specified form in a file called `stub.cfg` (stub configuration file). It is also in that file that the services that send and receive data to/from other objects must be described [15].
Every object has its own `stub.cfg` file. The information from this file will be used by the compiler when building the binaries. The file must be placed in the same directory as the C++ object program.
The stub configuration file begins by a line describing the name of the object. The next two lines declare how many subjects and observers the object has. Then, each service is described on a line. A service has a unique name in order to distinguish that service from other services in the system. One can connect the subject's service to the observer's service of another object by describing both service names in the file `connect.cfg` (see next paragraph).

Here is an example of an `stub.cfg` file:

```
ObjectName : SampleObject
NumOfOSubject  : 1
NumOfOObserver : 2
```

```
Service : "SampleObject.Func1.char.S", null, Ready()
Service : "SampleObject.Func2.int.O", Connect(), Notify()
Service : "SampleObject.Func3.Data.O", null, Control()
Extra : UpdatePowerStatus()
```

Lines starting by `Service` are the ones that describe the information of the gates. For example, in the line

```
Service : "SampleObject.Func1.char.S", null, Ready()
```

`SampleClass` is the name of the current object, `Func1` is the name of the gate the message will go through, `char` is the type of message been interchanged, and `S` means it is a subject (i.e. an outgoing gate); `O` would have meant it is an observer. 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 not usefull so it is set to `null`) and the second one is called when an AssertReady or a message is received [16] [6].

Description of the `connect.cfg` file

Now, in order to interconnect the objects, each subject must be assigned to one observer and each observer to a subject. The config file doing this is the `connect.cfg` file which has to be in the `OPEN-R/MW/CONF/` directory of the Aibo programming Memory Stick. When Aibo boots up, the system loads the objects and interconnects them according to the information in that file. There is only one `connect.cfg` file per program.

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

```
NameOfObject.NameOfSubject.MessageType.S
```

and ends by the corresponding observer, like:

```
NameOfObject.NameOfObserver.MessageType.O.
```

Of course, the type of messages exchanged by the subject and the observer must be the same at each side. For bi-directional connections, the configuration file must contain two lines, one for each direction of the connection. The definition of each part of the line follows the same specification as for the `stub.cfg` file described before [6] [15] [16].

The `Notify()` function

The `Notify(const ONotifyEvent& event)` is a special function that is called when a message arrives in the gate of the object where it is in. It is the only way to retrieve the message that was sent to the object. So, it is the entry point that one must use in order to get the data from a sensor or from

an input device like the camera. The message in question is passed in the function's argument each time that such a message is ready. Its content can be retrieved by casting the variable in which it was copied in (in our case it is the variable `event`):

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

The member functions receiving a message have to be described in `stub.cfg`, but it is not necessary to describe the member functions sending a message in `stub.cfg`. At the end of the `Notify()` function, an AssertReady must be sent to the subject that sent the message with:

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

[6] [15]

Aibo's time measurement

Time in Aibo's hardware is divided in discrete timesteps called *frames*. A frame is the smallest possible unit of time and represents 8 milliseconds [16]. So, there are 125 time steps in one second.

2.1.2 Aibo's camera

Accessing the camera

Sensors and joints are called *primitives* in Sony's official documentation. In Aibo's design, each primitive can be referred to by using a primitive locator supplied in the Sony's model information document [14]. The primitive locator provides the "address" of the primitive and the `OPENR::OpenPrimitive` static function convert this adress to an ID. In OPEN-R SDK the type OprimitiveID holds ID information. This design was chosen by Sony's developers in order to make objects portable between different Aibo models since the same sensor can have a different index within two different models [6].

Format type of the data sent by the camera

`OFbkImageVectorData` is the data structure that holds image data. It is the type of the data sent from the camera, i.e. the type of the messages sent from the outgoing gate named `FbkImageSensor` of `OVirtualRobotComm` [13]. Actually, a `OFbkImageVectorData` message contains the same picture in different resolutions but all in colour that are stored in different layers accessible by their indices. The index of the layer can be one of the following predefined constants: `ofbkimageLAYER_H` (high resolution), `ofbkimageLAYER_M` (medium resolution), `ofbkimageLAYER_L` (low resolution).

colour 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) [6].
The OPEN-R SDK provides a C++ class that handles image data called `OFbkImage`.

2.2 Webots

Even if the typical client for which the communication with the RCServer was designed is the remote control provided by the Aibo ERS-7 model in Webots, any other client can be used. The communication just uses port 54321 over TCP/IP. So that the RCServer understands the sent commands, they must respect the rules of the defined protocol as mentioned in section 3.4.
Only one connection is possible at one time. This does not include the wireless console which can always be used to see what Aibo prints out (the wireless console is accessible by connecting via telnet to the port 59000).

Actually, at the end of the development, a new version of Webots was not available. Hence, a small TCP/IP client was used for testing the new functionalities.

2.3 Protocol

For their interaction, the RCServer software runing on Aibo and the remote control in Webots on the client computer use a protocol explained in Lukas Hohls semester project report [4]. The messages used for sending commands to the RCServer are of a type called `BasicGetCommand` declared in the file `command_structure.h`. A `BasicGetCommand` is a simple structure containing two fields, a single character representing the command and another single character that can be used for example to identify a joint.

```
struct BasicGetCommand {
  char            command;
  unsigned char   identifier;
};
```

Actually, when a `BasicGetCommand` is sent to the RCServer, it arrives there as just two consecutive characters. So, in a simpler client, it is enough to just send two characters.

Chapter 3

Implementation

3.1 Image taking functions

The program code for the picture taking functionalities to the RCServer was inspired by the W3AIBO example of the samples provided by Sony [10]. The files `write_jpeg.c`, `write_jpeg.h` and `jpeg_mem_dest.c` were taken from this sample and directly included in the RCServer's source code directory and dependency list without applying any changes to them. The program `write_jpeg.c` contains two functions; first, `write_jpeg_mem` that as its name suggests, writes a JPEG picture to the random access memory (RAM) and returns the file size and secondly, `write_jpeg_file` that writes a JPEG picture to a file. Actually, for this project, only the function for writing the picture to the RAM was used since no function for writing a picture to a file on the Memory Stick was implemented. But the function `write_jpeg_file` was not removed on purpose since the RCServer could easily be extended in order to have such functionalities as proposed in section 4.1.
In order to be able to use these functionalities, three functions that were adapted from the `JPEGEncoder.c` program of the W3AIBO example were included in the RCServer: `GetJPEG`, `ConvertYCbCr` and `SaveJPEG`. Here is a description of what these functions do:

`GetJPEG` puts a JPEG file into the RAM. For this purposes, it uses the function `ConvertYCbCr`.

`ConvertYCbCr` combines the separated bands to a raw image and converts it to the JPEG file format. The pointer that was passed to the function as argument will then point to the beginning of the newly created file.

`SaveJPEG` can be used as a replacement for `GetJPEG`. This function saves a JPEG file to a file on the Memory Stick. It also uses the function `ConvertYCbCr` for this purpose.

3.2 Inclusions

In order to be able to compress the images to the JPEG file format, the jpeglib had to be included. Actually, an archive including the whole library was added to the program's directory. This is the same archive as the one provided on the OPEN-R homepage in the download section [9]. As its content must be compiled into the RCServer, the jpeglib had to be added to the dependency list of its Makefile, as well as the `write_jpeg` and the `jpeg_mem` functions described in section 3.1.

3.2.1 `stub.cfg`

In order to be able to get messages from the camera, a supplementary incoming gate had to be added in the RCServer's stub configuration file:

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

This means that we define an incoming gate (i.e. the RCServer object is an observer because of the `.O`) called `Image` that accepts messages of the type `OFbkImageVectorData` (the data type sent by the camera). Furthermore, the `Notify()` function is called each time this gate becomes a message (see section 2.1.1 for details about the `stub.cfg` file and for an explanation about the `Notify()` function and section 2.1.2 for details about the `OFbkImageVectorData` file format).

3.2.2 `connect.cfg`

An entry had to be added to the `connect.cfg` file in order to get data from the camera:

```
OVirtualRobotComm.FbkImageSensor.OFbkImageVectorData.S
RCServer.Image.OFbkImageVectorData.O
```

That means that messages of the type `OFbkImageVectorData` (the data type sent by the camera) are sent from the virtual object `OVirtualRobotComm` that is the hardware interface for the sensors, including the camera, through its `FbkImageSensor` gate to the RCServer object through its `Image` gate.

3.3 Image taking and sending

3.3.1 The `Notify()` function

Actually, the part of the RCServer where the most important part of code for image taking lies in the `Notify()` function. As explained in section 2.1.1, this function is called each time a message from the subject to which the object is connected to, is ready. This message is passed to the object through

the parameter of the `Notify()` function.

For the camera, a picture is ready all 4 frames, i.e. every 32ms. It is in this function that all the work for the camera support is done: an image is taken, converted to JPEG and then saved into the RAM. If that part is successfull, the header of the file is composed (see section 3.4.1), the image is copied to the shared memory buffer (see section 3.3.4) and finally the header and the image itself are sent at once to the connected client.

Since this function is called each time an image is ready, it is not possible to execute all the code for image taking every time because images are certainly not needed all the time and this would use the robot's resources unnecessarily. That's why all the code of the `Notify()` function is embedded in a conditional test checking the state of the camera. The camera's state is represented by a state variable explained in section 3.3.2

3.3.2 State variable

The state variable was implemented in the RCServer's header file as a simple enumeration structure called `ImageObserverState`. This was done in order to allow only specific values:

```
// states in which the camera exists
enum ImageObserverState {
    IOS_IDLE,        // do nothing
    IOS_PICTURE,     // take a picture
    IOS_CONTINUOUS, // send stream
    IOS_STOPSEND     // stop sending stream
};
```

The different possible states in which the camera can be are explained in the comments. Figure 3.1 contains an illustration of the finite state machine representing the states and transitions from one state to the other.

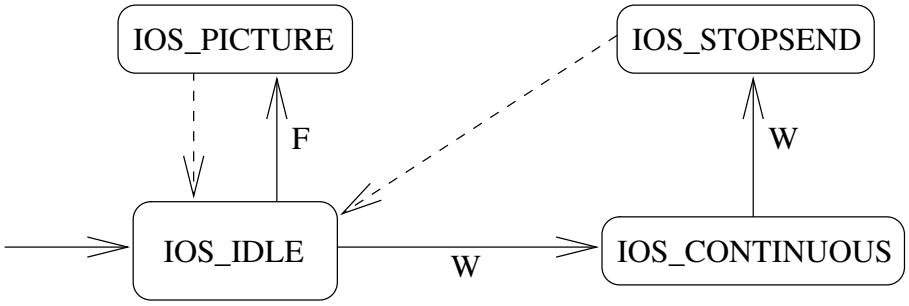

Figure 3.1: FSM representing the possible states of the camera

The dashed lines mean that in this state, the `Notify()` function will be executed only once and that the transition will be done unconditionally after that. Here is an explanation of the transitions:

- RCServer puts the `imageObserverState` variable to `IOS_IDLE` at start-up (initial state)
- when the RCServer receives the command `F`, it will make the camera go into a picture taking state, i.e. it will execute the code in the `Notify()` function once and then immediately go back to the idle state after that
- when a `W` command is received, RCServer executes `Notify()` i.e. the image-taking command as often as possible until it gets the same command again. Then, it will go into a transitional state and thereafter immediately return to the idle state.

The `IOS_STOPSEND` is a transitional state into which the camera goes just after receiving the command to stop sending a continuous video stream. This state is useful in order to be able to execute some special code after having sent the last picture of the stream, like e.g. closing the connection.

3.3.3 Image size

With the ERS-7 model of Aibo, the resolution of an image taken by the camera with the highest possible resolution (`ofbkimageLAYER_H`, see section 2.1.2) is 208×160. This is the resolution chosen for the implementation. It was decided not to support image sending in a smaller resolution since even if it is called *high resolution*, it is still small and should never slow down the communication because of its size. A buffer called `JPEG_BUFSIZE` of a size of 64 kilobytes is allocated in order to hold the compressed picture. This size of 64kb was simply chosen because a JPEG picture in that resolution should never exceed that size, even in its best quality.
The file size of a raw picture in the highest resolution is constant with 98 kilobytes. To be exact, it has a size of 99894 bytes. This value comes from the computation width $\times$ height $\times$ depth + header. So, here it is

$$208 \times 160 \times 3 + 54 = 99894$$

Once compressed with the standard quality of 85%, the file size lies between 3 and 6 kilobytes, depending on the content of the picture.

3.3.4 Sending of the image

In order to send data over the network, a shared buffer must be created in Aibo's memory. The data that will be sent has to be copied into that buffer and then will be copied to the protocol stack [12]. A shared buffer

Figure 3.2: Example of an image taken by Aibo

was already present in the initial implementation of the RCServer. It was only made bigger in order that an image will always fit into it. So, it was declared to have the size of `JPEG_BUFSIZE`.

3.3.5 Image compression

It was decided to compress the image from raw to the JPEG format on the Aibo, i.e. on the server side. It would also have been possible to make that compression on the client side. This decision was made since the file size of a JPEG image, compressed with the standard quality, is several times smaller than uncompressed. Since the compression on Aibo is very fast, the transfer will be much faster, which is very appreciated, especially when sending a continuous video stream.
Tests were made to know how long the compression and the sending takes. This information is retrievable by printing out the actual frame number (see section 2.1.1) each time the `Notify()` function is called. The results are listed in Table 3.1

	in frames	in milliseconds
only compression	4-8	32-64
compression and sending	8-25	64-200 (and more)

Table 3.1: Time needed for image taking and sending

It clearly appears that compression is very fast, sometimes even instantaneous (remember that the camera has a new image ready all 4 frames, see section 3.3.1) whereas sending can take a very long time.

3.3.6 JPEG compression quality

The JPEG file format provides several compression levels. A higher compression rate will produce a file of a smaller size but with more information losses. Conversely, a lower compression rate will produce a bigger file with less information loss. The different possible compression rates are called *quality* and vary from 0% to 100%, where the default is 85%. A quality of 0% shows the most artefacts and a quality of 100% represents the highest reachable quality of a JPEG file and does not mean that the image is not compressed at all.

The compression rate which the RCServer uses for the image it sends can be defined by the client using the identifier field of the `BasicGetCommand` it sends. Even if the type of that field is `unsigned char`, it is just interpreted as a number between 0 and 100. If it is over 100, it will be interpreted as 100 anyhow.

3.4 Extension of the existing protocol

In order to have some commands for controlling the camera, it was necessary to extend the existing protocol with new commands. By convention, a command that is sent to the RCServer is a capital letter. Actually, only two new commands were necessary; one for taking a single picture and one for beginning to send a continuous video stream. The command for telling RCServer to stop sending the stream is the same as the one used for beginning to send it. It is just sent once again. Two characters from the alphabet that were not already used were defined for that task. These commands are listed in Table 3.2.

BasicGetCommand		Description
char	uchar	
command	identifier	
F	quality	return a single JPEG file
W	quality	return JPEG files continuously

Table 3.2: Extension of the protocol for camera support

The answer of RCServer to each of these commands is the taken image preceded by a header whose purpose is explained in section 3.4.1.
In case the command for taking a single picture was sent, RCServer's answer will be only one image preceded by its header. In case the command for taking a video stream was sent, several pictures will be returned, each preceded by its corresponding header. The video stream is simply composed

of consecutively sent pictures. The frequency of images sent is the best that is technically possible. In application, a frequency of about 15 pictures per second is achievable. This can vary a lot as it mainly depends on Aibo's available resources and on the quality of the network communication. If Aibo is running objects that require a lot of computational power or if the network is saturated, the number of pictures sent per time unit will decrease. If this happens, the video stream will be more jerky than under ideal conditions and the stream can even be frozen temporarily.

3.4.1 Image header

As said before, when the RCServer sends an image, it does not just send the image, it first sends a header containing the letter `f` and then the size of the image file in bytes coded in big-endian order and only after this header the image itself is sent as shown in Table 3.3. This header is necessery for the client runing on the computer to know how many bytes to receive.

`f`	file size coded on 4 bytes	image data ...

Table 3.3: Format of an answer on an image taking command

Header and image are sent together at once from the RCServer. On the client side, the five bytes that contain the header must first be received in order to compute the file size of the picture before the picture itself can be received. Eventually, a check can be performed on the first byte in order to know if the header really does begin with the character `F`. If it does not, the operation can be aborted. If it does, the integer must be recomposed and converted to little-endian again.

One thing that is important to notice is that although RCServer usually sends the same character as the command character but as a small letter, for the sending of a video stream, it will still be the letter `f` that will first be sent since a video stream is nothing more than many single pictures shown continuously one after the other.

Chapter 4

Conclusion

4.1 Possible extensions

Communication over TCP/IP is synchronous, so it might not always be very fast. Since for a video stream it does not matter if all packets are not correctly received, it might be interesting to send it using UDP instead of TCP/IP. The UDP protocol is connectionless, i.e. packets are just sent without verifying if their are correctly and completly received. This allows a faster communication but does not guarantee in which order the packets are received. For the video stream, this means that theoretically it would be possible for an image that was taken after another one to be displayed before. In order to prevent this, we could number the pictures incrementaly and put this number in an extended version of the header. Then, on the client side, only pictures having a bigger frame number than the last picture that was shown would be displayed. Pictures with a small frame number would be ignored. Doing that, the video stream could eventually skip some pictures, but it could never happen that an older image would be displayed before a newer one.

It could also be interesting to give the user the possibility of saving a taken picture to a file on the Memory Stick. Actually, this was done at some point of the development but later removed given that the primary goal was to have the quickest possible image transfer, and also because writing to the Memory Stick is very slow in comparision to RAM access. Anyhow, the functions are still present in the RCServer's source code but are unused. Hence, having image saving functionalities could be easily achieved. The help functions for doing this are already included as explained in section 3.

A last thing that could be implemented for an optimal camera support could be to offer the user the possibility to control the camera settings. The `OPENR::ControlPrimitive()` can be used to change the gain, colour bal-

ance and shutter speed of the camera [6].

As the icing on the cake, one could imagine providing the with the possibility of seeing if the robot has detected a given colour since Aibo has a colour detection algorithm built in. This algorithm works very fast because it is encoded in hardware.

Bibliography

[1] Cyberbotics Ltd. *Webots Reference Manual.*

[2] Cyberbotics Ltd. *Webots User Guide.*

[3] Lukas Hohl. Aibo simulation in webots and controller transfer to aibo robot, June 2004.

[4] Lukas Hohl. Wireless remote control and monitoring of an aibo robot, February 2004.

[5] Sergei Poskriakov. Sony Aibo ERS-series robots support in Webots, 2004.

[6] François Serra and Jean-Christophe Baillie. Aibo programming using OPEN-R SDK tutorial, 2003.

[7] Sony Corporation. Official Sony OPEN-R internet page. `http://openr.aibo.com`.

[8] Sony Corporation. *OPEN-R SDK documentation English.* File `OPEN_R_SDK-docE-XXX.tar.gz`.

[9] Sony Corporation. Source of the JPEG library, file `jpegsrc.v6b.tar.gz`.

[10] Sony Corporation. OPEN-R sample programs, 2004.

[11] Sony Corporation. *OPEN-R SDK Installation Guide*, 2004.

[12] Sony Corporation. *OPEN-R SDK Internet Protocol Version4*, 2004.

[13] Sony Corporation. *OPEN-R SDK Level2 Reference Guide*, 2004.

[14] Sony Corporation. *OPEN-R SDK Model Information for ERS-7*, 2004.

[15] Sony Corporation. *OPEN-R SDK Programmer'sGuide*, 2004.

[16] Ricardo A. Téllez. Introduction to the aibo programming environment, 2005.