Winter Semester Project 2003/2004: Wireless Remote Control and Monitoring of an Aibo Robot

Lukas Hohl, semester 7

BIRG, Logic Systems Laboratory (LSL)
School of Computer and Communication Sciences
Swiss Federal Institute of Technology Lausanne

Supervision:
Prof. Auke Jan Ijspeert & Dr. Olivier Michel

February 12, 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Aibo is a four-legged robotic dog produced by Sony. Despite the fact that it is principally sold as an entertainment robot system, it has powerful capabilities such as wireless network communication, a wide range of input and output devices such as speaker and microphone, color camera, distance sensor, acceleration sensors, various touch sensors, LEDs and of course controllable joints.

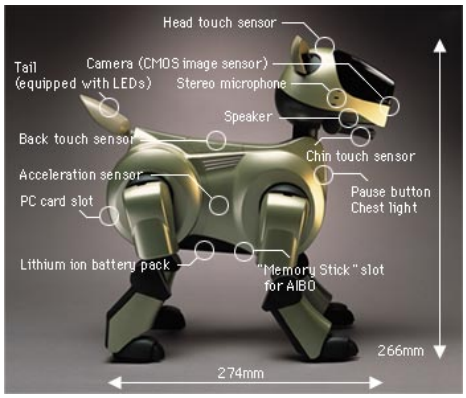

Figure 1.1: Aibo ERS-210 Overview

What makes the robot even more interesting, is the possibility to write custom programs for Aibo. The binary files are put onto a Memory Stick which is plugged into Aibo. For software development, Sony provides the OPEN-R SDK and a very complete documentation including sample programs.

The goal of the project was a standalone computer program that enables controlling and monitoring of an Aibo ERS-210(A) over a wireless network connection. This would allow the user to analyze the behavior of a real robot in comparison to a simulated Aibo. For this reason, a later integration into the Webots software (`http://www.cyberbotics.com/`) was already planned at the beginning of the project. The remote control software provides detailed control of all joints and has the capability to load files containing predefined movements (MTN files) onto Aibo and to play them back while observing the evolution of the real joint angles.

Chapter 2

System Overview

The Remote Control System is distributed on two hosts: An Aibo ERS-210(A) equipped with an ERA-201-D1 wireless LAN card and a PC with an IEEE 802.11b compliant wireless LAN card. Aibo runs a special OPEN-R software developed as one part of the project. The "Aibo Remote Control" software, which is the second part of the project, runs on the PC and establishes a wireless TCP/IP connection with its counterpart on Aibo. Aibo and the workstation exchange messages defined in the Communication Protocol. The protocol is the third achievement of the project. Figure 2.1 summarizes the global system architecture.

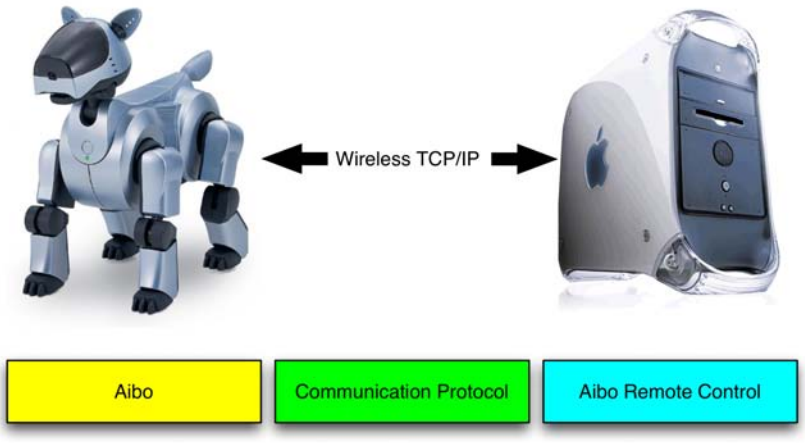

Figure 2.1: System Overview

Each of the three project parts is described in a separate chapter of this document. We start with the Communication Protocol, then we treat the software running on Aibo and finally the client's implementation will be explained.

Chapter 3

Communication Protocol

This chapter describes what needs to be known to interact with the remote control software on Aibo when the robot is considered as a "black box". Based only on the explanations given in the following sections, it should be possible to implement a client software which is able to take advantage of all the services provided by the remote control system. The descriptions do only reveal a minimum quantity of information about the implementation on Aibo, namely just enough to understand certain design choices.

The ERA-201-D1 wireless LAN card allows an Aibo ERS-210 to communicate with other hosts using the Internet Protocol (IP). IP version 4 (IPv4) is the version available on OPEN-R, Aibo's system software. The IPv4 protocol stack on OPEN-R includes several protocols that supplement the basic IP protocol[1]:

- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)
- DNS (Domain Name System)
- DHCP (Dynamic Host Configuration Protocol)

TCP was chosen for sending commands and receiving corresponding answers to and from Aibo. File transfers are also done using TCP. Aibo's default IP number is `10.0.1.100`. This number can be changed in a configuration file[2] on Aibo's Memory Stick. The port number on which Aibo accepts remote control commands was arbitrary fixed to `54321`. Only one remote control connection at once can be opened to Aibo. Further connection attempts do fail while the first connection is not closed. This does not affect `telnet` connections[3] on port `59000`.

The messages passed between Aibo and the controlling host are very much inspired by a command line interface. A character specifies the type of command. An identifier for the desired piece of information or the controlled robot component has to be provided. When a component is piloted, the desired value is also passed. Every command implies some kind of response. Aibo's answer is a repetition of the command character and the identifier plus the value that

[1] See [IP] for detailed information on supported protocols.
[2] See [Start] and [Install] for detailed instructions.
[3] See [Install] for more information and [Start] for performance notes.

was actually taken into account for the command or the current value of the read sensor. The answer is an error message if a problem was encountered on Aibo. In case of severe problems, Aibo closes the connection.

New commands are only read when the previous response was successfully sent back. Aibo does not send any data that is not requested by a command. It is necessary to do so, because all transmissions use the same network connection and both Aibo and the client need to know, what kind of data they can expect on the connection and what they are supposed to send. Thus the command protocol used by Aibo and the client is synchronous.

3.1 Access to robot components

There is an identifier for every physical Aibo component (and some other readable entities) that is controllable by the remote control software. The components (called primitives[4] in OPEN-R terminology) are given by name in the tables listing the identifiers for each category of primitives (tables 3.1, 3.2, 3.3 and 3.4) and other entities (table 3.5). Additional properties and restrictions are provided for some primitive types.

There must not be an intersection of the identifiers used for sensors (and other readable entities) and joints because every joint is also a sensor. LEDs, plungers and joints/sensors, however, are allowed to use the same identifiers.

Name	*ledID*
Eye light (Lower left)	1
Eye light (Middle left)	2
Eye light (Upper left)	3
Eye light (Lower right)	4
Eye light (Middle right)	5
Eye light (Upper right)	6
Mode indicator	7
Tail light (Blue)	8
Tail light (Orange)	9

LEDs can be either ON or OFF.

Table 3.1: LED identifiers

Name	*plungerID*
Left ear	1
Right ear	2

Plungers (joints with two possible positions) can be either ON or OFF.

Table 3.2: Plunger identifiers

[4]See [Model] for a complete list of primitives and robot schemas.

Identification		Position [degs]			Speed [degs/s]
Name	*jointID*	min	max	init	max
Neck tilt	1	-82.00	43.00	43.00	172.5
Neck pan	2	-89.60	89.60	0.00	172.50
Neck roll	3	-29.00	29.00	0.00	172.50
Mouth	4	-47.00	-3.00	-3.00	250.50
Left fore leg J1	11	-117.00	117.00	117.00	161.25
Left fore leg J2	12	-11.00	89.00	89.00	143.125
Left fore leg J3	13	-27.00	147.00	30.00	162.5
Left hind leg J1	21	-117.00	117.00	-117.00	161.25
Left hind leg J2	22	-11.00	89.00	70.00	143.125
Left hind leg J3	23	-27.00	147.00	30.00	162.5
Right fore leg J1	31	-117.00	117.00	117.00	161.25
Right fore leg J2	32	-11.00	89.00	89.00	143.125
Right fore leg J3	33	-27.00	147.00	30.00	162.5
Right hind leg J1	41	-117.00	117.00	-117.00	161.25
Right hind leg J2	42	-11.00	89.00	70.00	143.125
Right hind leg J3	43	-27.00	147.00	30.00	162.50
Tail pan	51	-22.00	22.00	0.00	256.25
Tail tilt	52	-22.00	22.00	0.00	256.25

Minimum speed for all joints: 12.50 degs/s.
Minimum acceleration for all joints: 15.63 m/s^2.
Maximum acceleration for all joints: 312.50 m/s^2.
Note: Speed and acceleration limits are absolute values,
they are valid for both directions of movement.

Table 3.3: Joint identifiers

Identification		Value		
Name	*sensorID*	min	max	Unit
Head sensor (back)	5	0	0.980665	N
Head sensor (front)	6	0	0.980665	N
Chin switch	7	0	1	binary
PSD (Position Sensing Device)	8	0	0.90	m
Left fore leg paw	14	0	1	binary
Left hind leg paw	24	0	1	binary
Right fore leg paw	34	0	1	binary
Right hind leg paw	44	0	1	binary
Thermo sensor	53	n/a	n/a	°C
Back sensor	54	0	1	binary
Acceleration y-axis	61	-19.6133	19.6133	m/s^2
Acceleration x-axis	62	-19.6133	19.6133	m/s^2
Acceleration z-axis	63	-19.6133	19.6133	m/s^2

Table 3.4: Sensor identifiers

Identification		Value		
Name	*miscID*	min	max	Unit
Remaining battery capacity	66	0	100	%
Battery temperature	67	0	227.00	°C
MTN key frame	99	-1	32767	integer

These identifiers don't have a corresponding robot primitive (physical component).

Table 3.5: Miscellaneous identifiers

3.2 Message types

The messages passed between Aibo and the client are implemented as C structures: `BasicGetCommand` is used to get a specific value returned from Aibo while `BasicSetCommand` allows the additional transmission of a value. The latter is the message type used for most commands.

Whenever a file must be handled, a `FileCommand` is sent to Aibo. This structure has fields for a filename, a flag and an option in addition to the command character. Aibo's answer to any kind of command is of type `BasicAnswer` in most cases. This structure makes possible the sending of a return value in addition to the repetition of the command character and the identifier. Error messages are also of this type.

Some commands might request multiple values to be returned at once. The structure `MultipleValueAnswer` is used for their transmission. No other data except the values is included in this message for efficiency reasons. The message only contains an array of values. The message is more of a container for data than a real message in the command line analogy. It should be combined with other messages to allow for error handling and exchange of control data.

The message types briefly described in the last few paragraphs are listed at the top of the tables summarizing the allowed commands and their answers (tables 3.7, 3.9 and 3.8). These tables also contain the types and names of all message structure fields. Type `char` corresponds to one byte, whereas `short` is two bytes long. `u` in front of the type means "unsigned". The units of the transmitted physical quantities are provided in a separate table (table 3.6).

3.3 Accepted commands

This section lists all remote control commands accepted by Aibo and gives a short description of the actions the robot executes in response. Tables show the different message types, the types and names of their fields and the content these fields are allowed to have (or must have) when a certain reaction is intended (tables 3.7, 3.9 and 3.8).

We start by giving a table of all physical quantities that have to be transferred to and from Aibo and their units (table 3.6). `short` was chosen as the type for value transmissions. This type is a good compromise between size in memory and precision. All non-binary values are multiplied by 100 before their transmission and again divided by 100 after their reception. The approach allows us to transmit floating point values with a precision of 10^{-2} in the interval

between -327.68 and +327.67. The tables in the previous section show that this is highly sufficient for all values that might be encountered.

Aibo's CPU is a little-endian MIPS processor. The two bytes of a `short` type are received in little-endian order from Aibo (contrary to the usual big-endian byte order in network traffic). No byte order inversion is made on Aibo. For the same reason, data sent to Aibo must be little-endian, too. This has to be considered when data is transferred to and from systems working in big-endian mode. On little-endian systems no special measures are necessary.

Physical quantity	Unit (precision)
Angular Position	10^{-2} degs
Angular Velocity	10^{-2} degs/s
Angular Acceleration	10^{-2} degs/s^2
Distance	10^{-2} m
Acceleration	10^{-2} m/s^2
Force	10^{-2} N
Temperature	10^{-2} °C
Percentage	10^{-2} %
ON / OFF	1 / 0
Counter	integer

Table 3.6: Physical units

The following subsections are supplementary explanations for the tables 3.7, 3.9 and 3.8. They should be consulted after studying the tables.

3.3.1 Reading multiple values

As table 3.7 states, the command for reading multiple values at once is `S 0`. The answer `s 0 0` does, however, not follow the command immediately. Between the command and the `BasicAnswer`, there is a `MultipleValueAnswer` message sent by Aibo. The message is an array of type `short`. The number of elements and what they contain is determined by the previously given commands to turn the observation of readable entities on or off. These commands are explained in table 3.9. `MultipleValueAnswer` contains an entry for every sensor or other readable entity whose observation was previously turned on.

The order of the values in the array is the following: joints, binary sensors, other sensors, misc (see table 3.10). The order of the primitives in one of the categories is the same as in the tables 3.3, 3.4 and 3.5.

The command `S 0 1` turns on (`S 0 0` turns off) the observation of all sensors and readable entities, this means all the sensors listed in the table 3.4 and the two first entries of table 3.5. The MTN key frame is not turned on (or off) by this command. Turning on the observation of the MTN key frame adds multiple values at the end of the array and, moreover, has effects on the rest of the message. A detailed description is given in section 3.4.

BasicGetCommand		BasicAnswer			Description
char	uchar	char	uchar	short	
command	identifier	command	identifier	value	
J	jointID	j	jointID	Angular Position	current position of a joint
S	jointID	s			
V	jointID	v	jointID	Angular Velocity	current upper speed limit of a joint
A	jointID	a		Angular Acceleration	current upper acceleration limit of a joint
S	sensorID	s	sensorID	various	value of any kind of sensor or other readable entity
	miscID		miscID		
	0		0	0	value of every observed entity (see 3.3.1)

Table 3.7: Messages of type BasicGetCommand

FileCommand				BasicAnswer			Description
char	uchar	char[12]	ushort	char	uchar	short	
command	flag	filename	option	command	identifier	value	
U	ignored	(see 3.3.2)	filesize	u	0	0	upload a file to Aibo (see 3.3.3)
D	ignored	(see 3.3.2)	ignored	d	0	0	delete a previously uploaded file (see 3.3.3)
P	(see 3.4)	(see 3.3.2)	loops	p	0	loops	play back an MTN file (see 3.4)

Table 3.8: Messages of type FileCommand

BasicSetCommand			BasicAnswer			Description
char	uchar	short	char	uchar	short	
command	identifier	value	command	identifier	value	
S	0 sensorID miscID	zero / non-zero	s	0 sensorID miscID	OFF / ON	turns off/on observation of all sensors (see 3.3.1) turns off/on observation of a sensor (see 3.3.1) turns off/on observation of an entity (see 3.3.1)
J	0 jointID	Angular Position	j	0 jointID	Angular Position	sets goal position of all joints (see 3.3.5) sets goal position of a joint (see 3.3.4)
V	0 jointID	Angular Velocity	v	0 jointID	Angular Velocity	sets max. speed of all joints (see 3.3.5) sets max. speed of a joint (see 3.3.4)
A	0 jointID	Angular Acceleration	a	0 jointID	Angular Acceleration	sets max. acceleration of all joints (see 3.3.5) sets max. acceleration of a joint (see 3.3.4)
L	0 ledID	zero / non-zero	l	0 ledID	OFF / ON	sets state of all LEDs sets state of a LED
K	0 plungerID	zero / non-zero	k	0 plungerID	OFF / ON	sets state of all plungers sets state of a plunger
Q R	ignored	ignored	q r	0	0	immediate shutdown immediate reboot

Table 3.9: Messages of type BasicSetCommand

3.3.2 Restrictions on filenames

A filename can be at most 12 characters long because on Aibo filenames must have the 8+3 format. The base name and the extension must be separated by a period character (`.`). Lowercase letters are automatically converted to uppercase letters by Aibo's system software.

3.3.3 File handling

The size of an uploaded file has to be specified in the upload command. The size is given as the number of packets necessary for the transmission minus one (the first packet does not count). The last packet might not be completely used, but it has to be counted, too. This allows the transmission of files with a size of up to 2^{16} times the packet size (the size is transmitted in the option field, which has type `ushort`). The packet size used in the current implementation is 512 bytes. It results a theoretical file size that is twice as big as the capacity of an empty Aibo Memory Stick (16 MB).

Right after the upload command, Aibo awaits the number of packets given in the command. Any data can be transferred in the packets, as long as the file content is exactly mapped to the packets. The last packet doesn't need to be be completely filled. All the uploaded files are put into a special directory on Aibo's Memory Stick. The location can not be specified. Any already existing file with the same name is overwritten. When Aibo has received the exact number of packets, it sends back the answer as it is specified in table 3.8. A started upload must be completed by sending as many packets as the upload command stated.

A file that has to be deleted on the Memory Stick is specified by its name. If the file does not exist (anymore), an according error message is returned (see 3.3.6).

3.3.4 Commands for one joint

Table 3.3 lists various limitations (either specific to a joint or global) for the position, speed and acceleration of a joint. No indication on speed and acceleration limits is given in [Docs]. In [Notes] there is a direct link to a [FAQ] entry where at least upper angular speed limits for Aibo's joints are listed. The lower speed and both acceleration limits were chosen to maximize consistency, i.e. the combination of the minimum speed and the maximum acceleration still results in a non-zero acceleration duration (with respect to the time units on Aibo).

The answer to a command setting a position, speed or acceleration value does always contain the value that was actually taken into account. This might not be the commanded value. Thus if the desired value surpasses the maximum position, speed or acceleration respectively, the answer states the maximum value allowed. Accordingly, the same is true for the minimum. Desired velocity and acceleration limits must be transmitted as absolute values, negative values are considered as being smaller than the minimum.

3.3.5 Commands for multiple joints

When the position, speed or acceleration is set for all joints at once, Aibo treats every single joint as described in 3.3.4. However, the returned value in the

answer message does not reflect any of this limitations and is just a copy of the commanded value.

3.3.6 Error handling

If Aibo does not understand a command, either because it is not of any of the required types or because its content doesn't make sense, the robot returns a `BasicAnswer` with the command field equal to 'e' and the other two fields equal to '0' (`e 0 0`). Depending on the command given, this allows to track down most error causes. In most of the cases, either the command character is wrong or the provided identifier does not exist. Depending on the implementation of the client software on the controlling host, a message type inappropriate for the given situation or a message not corresponding to any of the required types might also be the cause.

When handling files, another error message might occur: `e 0 1` (value field equal to '1'). This error is the result of either a delete or a playback command specifying a file that does not exist on Aibo's Memory Stick. If a file for playback exists but does not have a `.MTN` extension, the value field of the error message is equal to '2' (`e 0 1`).

If a runtime problem not caused by a wrong command is encountered, the network connection is closed. A client software should treat this case correctly, i.e. define timeouts for send and receive operations.

3.4 MTN files

MTN files can contain joint positions and other command data over a certain number of so called key frames (see section A.1 for the specification of the MTN file format). In the current MTN version, a key frame always has a duration of 16 ms. The remote control system allows the upload and playback of MTN files (see commands in table 3.8).

In the current implementation, an MTN file can only contain position data for the three neck joints and the twelve leg joints (3 per leg). Other files will simply not be executed, there is no error message. See 3.3.6 for possible error causes.

The option field of `FileCommand` is used to specify the number of times the file given by the filename field will be executed. Aibo moves to the position corresponding to the first key frame before it actually starts executing the file. The same is done with the last key frame in cases it was not exactly reached after the last loop. Position commands are ignored while executing a file, but for instance specifying zero as the number of loops in a subsequent playback command stops the MTN execution after the current iteration.

When the observation of the MTN key frame is turned on (see command in table 3.9), a certain number of modifications are made in the message returned when all sensor values are requested (see 3.3.1). After the last sensor data entry, the current MTN key frame is transmitted. This is either a positive integer telling at which key frame of the MTN file the measures were made or it is equal to '-1' to tell that no MTN data is currently executed and that the values after the key frame entry do not have any reasonable meaning.

After the MTN key frame entry in `MultipleValueAnswer`, the fifteen joints mentioned at the beginning of this section each have an entry (independently of their current observation state and in the order given by table 3.3) with the position commanded by the MTN file for the key frame in question. Figure 3.10 shows the order of the values in the answer message. Moreover, if the key frame is valid, i.e. not equal to '-1', the measured positions of the fifteen MTN joints in the entries before the MTN key frame entry (if the joint is observed) reflect the positions at the exact time instant when the robot should have reached the commanded position. This allows direct comparison between the commanded and measured values in the same answer message. When the MTN key frame is not observed or not valid, the measures are made at the instant of the reception of the command requesting all sensor data.

0 to 18	0 to 6	0 to 7	0 to 2	0 or (1+15)	
joints	binary sensors	sensors	misc	MTN key frame	MTN joints

Table 3.10: Transmission of multiple values including MTN data

The flag field in `FileCommand` is used to specify how often the commanded position and the real position shall be measured. The number in flag divided by two gives the number of key frames between two measures (a frame on Aibo is half as long as an MTN key frame, an uneven value makes the rate constantly vary by one). This value should not be greater than '16' and at least equal to '2' (inappropriate values are corrected). The most up to date values are returned in `MultipleValueAnswer` when a command requests it. It is the client's task to query the sensor data at an appropriate frequency in order to not miss any key frame. Two requests in a short time interval might return the same measures.

3.5 Possible extensions

In the current implementation, not all fields are used for every command. Especially `FileCommand` has a lot of possibilities to add supplementary parameters for more complex applications. Also `BasicAnswer` is prepared to do more complex error handling (see 3.3.6). More messages could be added to the protocol.

There are many status parameters accessible on Aibo (overall status, battery, network) for which identifiers and appropriate units could be found in order to include them as miscellaneous sensors or to define proper commands for their reading. Sensor reading could even be done on a separate connection (where Aibo possibly sends data continuously without requests).

It would be interesting to include control over Aibo's speaker, microphone and camera. Such commands would certainly need more file transfers to and from Aibo, especially for sound and image files. Usage of the UDP protocol might provide advantages over TCP for such transfers.

More powerful file handling might become desirable. There is a sample programs for Aibo that implements a simple FTP server (`TinyFTPD` in [Samples]). Either FTP functionalities could be implemented as an extension to the protocol, or a dedicated FTP server could run on Aibo in parallel to the remote control system and act independently, i.e. have its own connections with different modules of the client software or even with external applications.

Chapter 4

Aibo

4.1 OPEN-R

"OPEN-R" is the interface that Sony is promoting for the entertainment robot systems to expand their capabilities.[1] The "OPEN-R SDK" discloses the specifications of the interface between the "system layer" and the "application layer" and makes the development of custom Aibo software in C++ possible.

Practical aspects of the OPEN-R SDK are discussed in [Start]. The present section treats the basic concepts of OPEN-R.[2] Features of OPEN-R:

- *Modularized software and inter-object communication*
  OPEN-R software is object-oriented and modular. Software modules are called "objects". Processing is performed by multiple objects with various functionality running concurrently and communicating via inter-object communication. Connections between objects are defined in an external description file. 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 (e.g. 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 functionality, without requiring detailed knowledge of the robot hardware. The system layer also provides the interface to the TCP/IP protocol stack, which enables the creation of wireless LAN networking applications. 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.

[1] Visit the web page [OPEN-R] to learn more.
[2] A more detailed discussion can be found in [Prog].

4.1.1 Object

OPEN-R application software consists of several OPEN-R objects. These are not objects in the common object-oriented way of thinking. The concept of an object is similar to one of a process in UNIX or Windows operating systems. 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. The file is put on an Aibo Programming Memory Stick. When the robot boots, the system software loads the file from the Memory Stick and executes it as an object.
- *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 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 (this is an infinite-loop, an object cannot terminate itself):
  1. Loaded by the system
  2. Wait for a message
  3. When a message arrives, execute the corresponding method. Possibly send some messages to other objects.
  4. When the method finishes execution, go to step 2.
- *An object has multiple entry points.*
  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.

4.1.2 Inter-object communication

The use of inter-object communication enables each object to be created separately and later be connected to other objects. This results in a very efficient development lifecycle. 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 that the observer is ready to receive data or not. If the observer is not ready to receive data, the subject should not send any data to the observer, otherwise messages might be ignored.

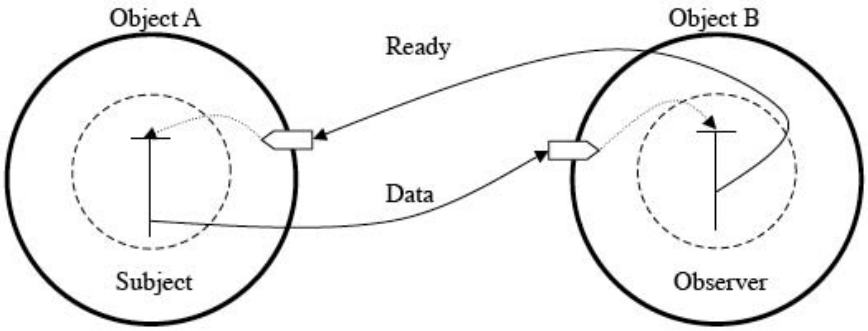

Figure 4.1: Inter-object communication

4.2 Remote Control

The remote control software running on Aibo consists of three OPEN-R objects. Figure 4.2 shows them as yellow circles. `PowerMonitor` is isolated from the other objects. `JointMover` and `RCServer`, however, have multiple entry points (in addition to the default entry points, which are not shown on the figure). The entry points are labeled with the service names. Red arrows show message passing between objects (the subject is at the beginning of the arrow and the observer is where the arrow points to). The arrows point in the direction of the notify events. A label shows the type of data transmitted, e.g. joint and LED commands need to be passed in the form of command vectors containing commands for multiple primitives and over a certain period of time. The ready events are passed in the opposite sense (not shown by a separate arrow, contrary to figure 4.1). `RCServer` possesses additional entry points necessary for the network communication. These entry points are not drawn individually.

A fourth object is shown on figure 4.2. `OVirtualRobotComm` is represented by a white circle. This object is part of Aibo's system software and provides services for sending commands to the robot, reading sensor values and retrieving images from the camera. The Effector service receives joint and LED commands.

The following paragraphs discuss in a more detailed manner the three objects forming the remote control software.[3]

4.2.1 PowerMonitor

This object is almost identical to the original sample program provided by Sony. The sample `PowerMonitor` is contained in [Samples]. Only a very small modification was made for the purpose of this project.

OPEN-R offers means for continuously observing changes in the power status of the robot. The power status includes the robot and battery status, the remaining battery capacity and its temperature and other status information. It can be precisely specified what parameter changes cause a notification (not

[3]Consult [Ref] and [IP] for implementation details.

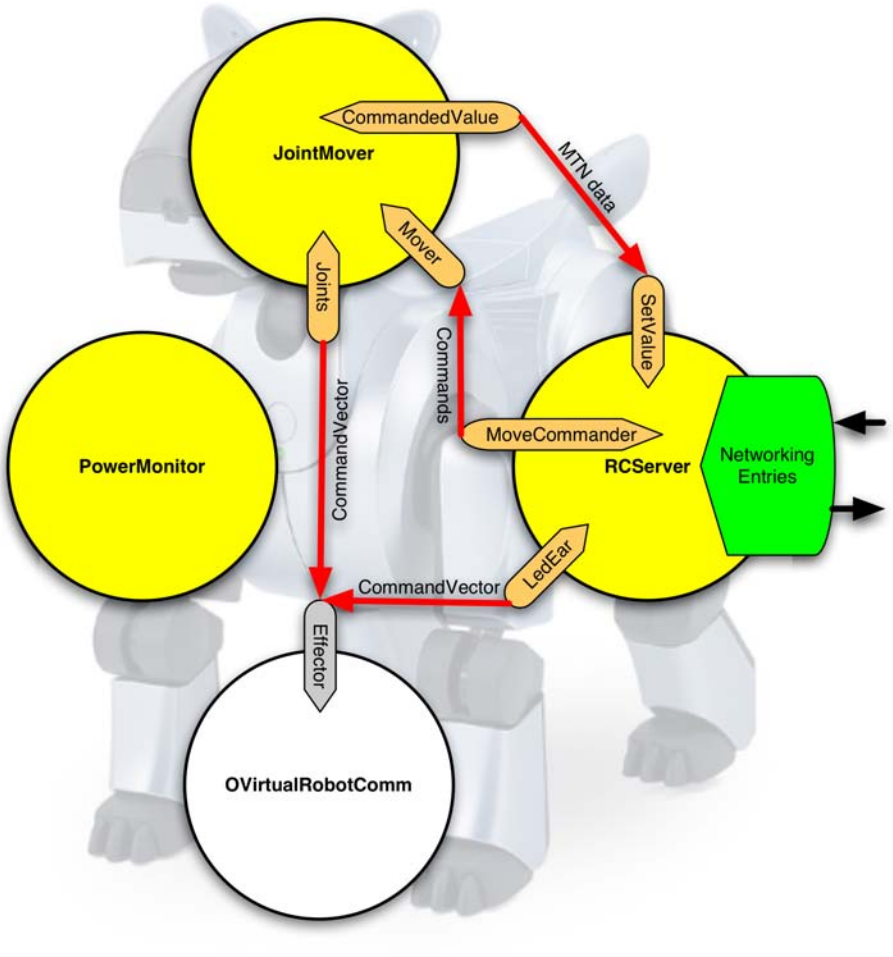

Figure 4.2: Objects on Aibo

possible with certain parameters) and on which entry point the notification shall be received.

PowerMonitor has a so called extra entry point, which is not shown on figure 4.2. It's this entry point that will be notified when a change in an observed component of the power status occurs. In PowerMonitor, every change in the robot and battery status is observed. Robot and battery status each consist of an array of bits that are set according to the current state. These bits are tested and actions are taken accordingly.

Aibo is immediately shut down when (the last point was added for the remote control software and is the only addition compared to the original version by Sony):

- The pause switch is on.
- The battery capacity is low (approximatively 20 %).
- Aibo is connected to an external connector. External connectors include connectors of the AC adaptor and the station.

No direct notification of the other two objects is made in case of a shutdown. As there are two methods common to every object, which are called by the OPEN-R system at shutdown, appropriate actions specific to a certain object can be done in its implementation of these methods, e.g. RCServer closes the network connection at shutdown.

4.2.2 RCServer

The RCServer object is the central part of the remote control software. It is the one that listens for TCP connection requests and then handles all the network traffic with the client. It interprets the commands received and performs the appropriate actions, either directly or by delegating them to specialized objects (JointMover). The answer to a command is generated and returned over the same network connection.

Networking and command interpretation

The networking implementation in RCServer is very similar to the one in the `EchoServer` sample program in [Samples]. The four basic networking operations are:

- Listen
- Receive
- Send
- Close

Every operation is implemented using two methods, an internal one and one that corresponds to an entry point. Thus every networking operation also corresponds to one entry point. These entry points are not shown individually on figure 4.2. The internal method passes a special message to the IPStack object and specifies the corresponding entry point and the attached method as the receiver of the answer. This method then calls the internal method corresponding

to the next networking operation, e.g. a successful initial Listen calls Receive, Receive calls Send and vice versa during normal operation and after reception of the answer to a Close operation, Listen is again called. The decomposition of every operation in two methods introduces asynchronous behavior in the sense that RCServer can accept calls on other (non networking) entry points while the IPStack is actually executing a networking operation. The communication protocol between Aibo and the client, however, is synchronous due to the ongoing switching between Receive and Send.

Right after the successful reception of a message over the network, Aibo analyzes the potential command and prepares a corresponding answer for the sending back over the network. Only after the complete interpretation (and execution) of the command, the Send operation is invoked. This means that the constant switching between Receive and Send is blocked just before the Send operation while a command is treated.

The analyze of the command first checks the size of the received message. If it does not correspond to a known type of command, an error message is returned. Then the command character is checked. Furthermore the identifier must be known. When these two checks fail, an error message is sent back, too. The value of a command cannot cause an error. If the value does not correspond to the needs, it is adapted according to the descriptions in the previous chapter. Syntactically correct commands for file handling, however, may cause errors depending on their actual content.

Some commands, such as the sending of all observed sensor data or the reception of an uploaded file, imply the sending or reception of data before the transmission of the normal answer message. The command messages for doing these sends respectively receives are passed to the IPStack without specifying an entry point. This means that the execution in RCServer is blocked while IPStack does the transmission of the data and that RCServer cannot do anything else until successful completion. This approach was chosen for the integration of the additional networking operations into the normal execution order which consisting of an ongoing switching between Receive and Send. This is consistent with the fact that every command implies an answer and that new commands are only read when the previous response was successfully sent.

Sensors

We treat here the execution of all the command listed in table 3.7. The correspondence between command identifier and the OPEN-R identifier of a robot component[4] is established by maps. This technique is also used for other situations where such a correspondence has to be made.

Single joint positions and sensor values are read by a direct call to the appropriate OPEN-R method with the desired OPEN-R identifier as a parameter. Unit transformations depending on the type of sensor are done before writing the value into the answer message. The remaining battery capacity and temperature are extracted from the robot power status, which is obtained by a direct OPEN-R method call, contrary to the constant observation done in PowerMonitor. For joint velocities and accelerations, there exist data structures in RCServer which contain the last commanded value (or the default value when

[4]OPEN-R identifiers are dynamically obtained by "opening" a primitive while providing its CPC locator given in [Model].

no previous command was given). It is this value that will be read and returned in response to a command requesting the velocity or acceleration limit of a joint.

Turning on or off the observation of a sensor (or all sensors) by a command from table 3.9 results in a corresponding boolean value (or multiple boolean values) being set in a map. When all the observed sensor values are requested, these booleans are tested and the values of the observed sensors are written to the multiple value message. The values are obtained as described in the previous paragraph. If the MTN key frame is observed, the last values (key frame index, commanded and real joint positions) received from JointMover over the SetValue entry point (see figure 4.2) are written to the message (real joint positions only if the corresponding joint is observed). The message is sent to the client before the normal answer.

LEDs and Plungers

Commands setting (see table 3.9) LED and plunger values are directly executed by RCServer, contrary to joint commands which are delegated to JointMover. Whenever a network connection is established by a client, the LEDs and plungers are commanded to take their initial positions (LEDs are off and ears are in top position, meaning on). Parts of the code are based on the `BlinkingLED` sample in [Samples].

Two command vectors are maintained in RCServer. They are identical, as they both contain an element for every LED and plunger. The minimum illumination duration for LEDs is 512 ms and cannot be changed. One of the vectors (in fact a reference to it) is always passed to OVirtualRobotComm (Effector entry point on figure 4.2). The other vector has the role of a buffer for commands coming in while Aibo is executing the passed vector. The buffer vector's elements are set according to the received commands (one or multiple element at a time, depending on the type of command) or according to the execution phase of the program, e.g. upon initialization.

The passed vector must not be modified before any previous execution is completed, because only a reference to the vector is passed. When Aibo is ready to accept a new command (i.e. it has finished the execution of the passed vector or it is the first command), a copy of the buffer vector is made into the vector intended for passing and then this last vector is given to OVirtualRobotComm for execution. Otherwise the fact that there were modifications in the buffer is memorized and a copy of the buffer is passed as soon as RCServer receives the ready notification by OVirtualRobotComm.

The same element in the buffer might have changed multiple times while the object is waiting for the ready notification. Only the current value at the instant when the ready notification is received is taken into account. The actual execution of the command vector by Aibo might happen after sending the answer message to the client.

Joints

RCServer analyzes valid (the identifier corresponds to an existing joint) joint commands (see table 3.9) to check if the values respect the limits listed in table 3.3. If necessary, values exceeding the maximum or minimum limits are adapted. A copy of the (modified) command is then passed to JointMover (Mover entry

point on figure 4.2). It is the adapted value that is returned over the network in the answer message (if the command concerns a single joint). When a new velocity or acceleration limit is set, RCServer keeps its own copy of the adapted value to enable easy reading of this value when it is requested by the client.

Commands for all joints are decomposed into individual commands which are then passed one by one to JointMover. This way JointMover does not have to do any checks on the given values, because they are all adapted in RCServer. A first (adapted) command is passed immediately to JointMover. It is kept track of how many joints still have to receive the command at what was the commanded value. Whenever RCServer receives the ready notification by JointMover, the next command is sent, until all joints have been treated. The commands can not be all sent immediately, because messages are simply deleted if the observer is notified when it is not ready. When a second multiple joint command arrives from the client, the process of giving commands to every single joint is interrupted and starts from the beginning. When there are commands setting positions, velocities and accelerations, priorities are given in this order, i.e. first all joints receive the position command, then the velocity command and finally the acceleration command (the only exception is the very first command which is passed right after the reception of the command message from the client).

In theory, it would be possible that a rapid sequence of many (multiple) joint commands causes messages to be deleted because JointMover can't treat the commands fast enough and is not ready when the notification is made. For commands of the same type all concerning the same joint this is no problem. In practice, JointMover treats the received messages much faster than commands can be given over the network. The problem must however be considered when JointMover is used in a different context and is given commands by other means, e.g by another object generating commands automatically and in short intervals of time.

Shutdown and Reboot

When such a command is received, Aibo turns off its motor power and then makes a call to the OPEN-R shutdown method. A boot condition has to be specified when doing so. The shutdown command sets this condition to "Starts with the pause button". This means that the robot will only boot again when its pause switch is pressed. The reboot command sets the condition to a timeout of zero relative to the current time. This causes Aibo to reboot immediately after shutdown.

The shutdown method is called before the answer is sent back to the client. Because the execution of this method takes some time, the response message is still transferred to the client which can then take the appropriate measures (i.e. close the connection or handle the connection closure by Aibo). Aibo itself closes the connection when the OPEN-R system calls the termination entry points of RCServer.

Files

All files are put into (and deleted from) `/MS/OPEN-R/MW/DATA/P/` on Aibo's Memory Stick. RCServer does all the writing (including deletion) on Aibo's

Memory Stick. Some of the standard functions for accessing the file system can be used on Aibo, but not all of these methods exist and some have a behavior specific to OPEN-R.

Files are deleted by RCServer when such a command is received. The response message might say that the file does not exist on Aibo. When uploading a file, RCServer tries to read the exact number of packets given in the command. The last packet doesn't need to be be completely filled with data. All the packets are written to the same file (which overwrites any already existing file with the same name). One single packet is read from the network and appended to the file in a method which calls itself recursively if there are any more packets to be received. Upon successful completion, the usual answer is sent to the client.

Any playback command is forwarded to JointMover (Mover entry point on figure 4.2) if it meets the following two conditions: The specified file exists and has a `*.MTN` filename extension. Otherwise an error message is returned to the client and JointMover does not receive any command.

4.2.3 JointMover

The JointMover object executes all the commands concerning joints (setting positions, velocities, accelerations and playback of MTN files). It receives them from RCServer on the Mover entry point (see figure 4.2). The commands are already adapted to respect joint limits (see table 3.3) and the MTN files are guaranteed to exist on the Memory Stick. The number of tests and modifications on the commands is thus significantly reduced. In fact, every command concerning files is considered to be a playback command and all the other commands are supposed to be setting commands for one single joint with a feasible value.

JointMover can be in different states. A simplified state diagram is shown on figure 4.3. The actual implementation has more states. They are used during initialization, e.g. to fill a first command vector with valid data and to set joint gains. After system startup, JointMover moves the joints to the initial position. The posture called "broadbase" by Sony was chosen. For this initial movement, the same algorithm as for all manually commanded movements is used. The only difference is, that the goal positions are set by the program itself. When Aibo has reached its initial posture, it starts accepting commands.

The state transitions are either done when a certain command is received (e.g. the transition from "Accepting Commands" to "Prepare MTN") or when a movement algorithm discovers that no more joints must be moved (e.g. "Prepare MTN" to "MTN", "MTN" to "Finish MTN", "Finish MTN" to "Accepting Commands"). The method invoked by the ready notification received from OVirtualRobotComm on the Joints entry point (see figure 4.2) tests the current state and does the corresponding action, i.e. invoke the right movement algorithm to continue the movement when not all joints have reached their goal positions or do the state transition. There is no transition done when a movement has finished in state "Accepting Commands". JointMover remains in this state as long as no MTN playback command arrives. In general, the first invocation of a movement algorithm is done when a corresponding command (individual joints or MTN) is received from RCServer (which receives the ready notification afterwards) and any further executions are caused by the ready method if necessary.

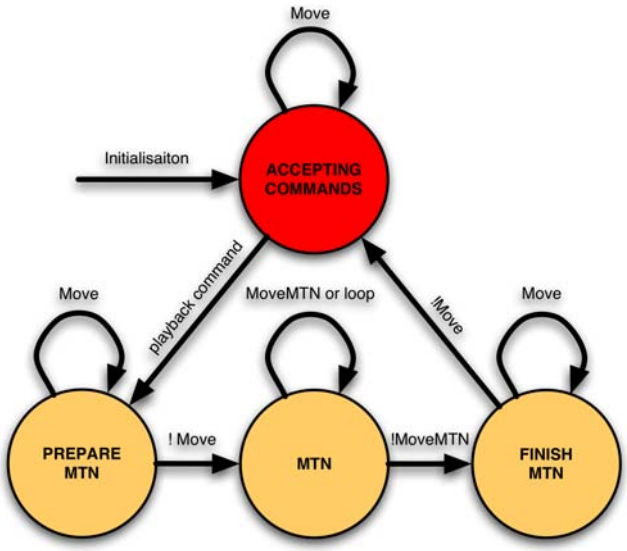

Figure 4.3: JointMover State Diagram

Individual Joints

New positions, velocity and acceleration limits are stored in persistent data structures as soon as such a command arrives. Position commands are ignored when JointMover is not in state "Accepting Commands". Velocity and acceleration commands are always accepted, but the new limit is only taken into account immediately when it is greater than the old one (recall that the limits are absolute values and are valid for both directions of movement). If the new limit is smaller, the old value is replaced when the joint in question is not moving anymore. For the acceleration limit, this prevents the situation, where it might become impossible for a joint to stop at the physical limit position because the new acceleration value is too small. For the velocity limit, the same is done for symmetry reasons.

The basic time unit for all commands given to Aibo is 8 ms. This is the duration of a so called "frame". A command vector can contain at most 16 frames per joint. Every frame contains a position value. It is thus natural to choose "frame" as the basic time unit.

After updating the goal position (or the velocity or acceleration limit), JointMover calculates the movement parameters. Every joint movement is composed of the three following phases and the result of the computations is the the duration of every phase:

- acceleration
- constant speed
- deceleration

The duration values are reevaluated every time a joint receives a new command. The calculation are always done at the instant of the joint command re-

ception (the only exceptions are mentioned in the first paragraph). This allows computation of new movement parameters while Aibo is moving, i.e. executing a previous command vector.

There is one command vector maintained in JointMover. This vector (in fact a reference to it) is passed to OVirtualRobotComm (Effector entry point on figure 4.2). The passed vector must not be modified before any previous execution is completed, because only a reference to the vector is passed. When Aibo is ready to accept a new command (i.e. it has finished the execution of the passed vector) and there are joints that have to be moved, the vector is filled with of up to 16 frames per joint and then passed to OVirtualRobotComm. Otherwise this is done as soon as JointMover receives the ready notification by OVirtualRobotComm. The construction of the command vector can thus happen either immediately after the calculation of the phase durations or when Aibo is ready again. Only 16 frames per joint can be packed into one vector and this is in most cases not sufficient to reach the goal position. The next vector is constructed in the method invoked by the ready notification received from OVirtualRobotComm on the Joints entry point (see figure 4.2). When there are no more joints to be moved and the command vector was not modified, it is not passed to Aibo. The following paragraphs describe the construction of the command vector, i.e. the movement algorithm.

Counting the number of executed frames since the last command allows to know in what phase the joint currently is and it is possible to fill position values into each frame of the command vector accordingly. The movement algorithm loops through all joints and all frames in the vector and, while filling in the position values, updates the (calculated) values for the current position and velocity of a joint. These "current" values are maintained in JointMover and are not the real ones at the time instant of the computation, but the values that are reached when Aibo will be executing the frame in question. Depending on how many frames there are left to finish a movement, only part of the vector might be used for a certain joint.

During the acceleration phase, the maximum value for the acceleration is always taken. The new speed is equal to the (calculated) speed for the previous frame plus the value of the acceleration limit (taking into account the current direction of movement). This speed is added to the current position and the resulting value is written into the command vector. A symmetric procedure is done for the deceleration phase. For the phase with constant speed, the current velocity is used unchanged to determine the new position. When a joint has finished its movement (i.e. the number of frames done corresponds to the sum of the three phases), the current position is explicitly set to the measured value. This prevents potential inaccuracies from being propagated on later movements. For the same reason, the speed is set to zero. At this moment, the velocity and acceleration limits are updated if any command specifying a smaller limit than the current one has arrived during the movement.

Aibo will in any case execute the vector as is, it must not be changed after the notification. The calculated current position and velocity of a joint contain the values corresponding to the last valid frame in the vector and are thus the values that will be reached after the successful execution of the vector. The values are used as the basis for further computations of phase durations (and later vector constructions). This allows the calculation of the movement parameters before the currently executed vector is treated completely. No measured values are

used except after a joint has reached its goal position.

The last thing that has to be explained is the computation of the phase durations. The minimum speed and the maximum acceleration limits (see table 3.3) will result in an acceleration phase with a duration of 5 frames. The decision not to accept smaller speed and acceleration limits while the joint is moving simplifies these calculations. Two basic cases are distinguished:

- The direction of movement has to be changed.
- The movement can continue in the same direction.

In the first case, the joint must come to a complete stop before the movement in the opposite direction can start. The number of frames necessary to stop will be added to the acceleration phase later. The distance necessary to stop is taken into account for the computation of the durations of the three phases of movement after the stop. When the new distance is long enough, the speed limit will be reached and there is a phase of constant speed between the two equally long phases of acceleration/deceleration. Otherwise there is an acceleration to reach an intermediate speed below the limit and an immediately following deceleration to zero speed. These two possibilities are similar to the two first sub-cases of the situation where the movement can continue in the same direction. In any case, the number of frames necessary to come to a stop is finally added to the acceleration phase. Because the intended direction has changed, the procedure applied in the movement algorithm (addition of the acceleration value to the current speed with respect to the direction of movement) will produce a soft turnaround.

The second case has three sub-cases:

1. The distance is long enough to reach maximum speed.
2. Only an intermediate speed below the limit can be reached.
3. The distance is too short to stop at the goal position.

The first sub-case is similar to the case where the direction of movement changes and the distance is long enough to reach full speed with the difference that the acceleration does not start at zero speed but at any velocity between zero and the limit. There might even be no acceleration phase because the joint is already moving at full speed.

In the second sub-case, there is no phase of constant speed. The computation of the intermediate speed is slightly more complicated than in the case where the direction of movement changes, because, like in the first sub-case, the joint might already have a current non-zero speed. The maximum speed reached between the current and the goal position is not necessarily higher than the current speed. Again, the intermediate speed might be equal to the current speed and there's only deceleration.

The third sub-case is very similar to the case where the direction of movement changes. The joint must in any case come to a complete stop before the other phases are computed. Here, however, we are sure that there is no phase of constant speed, because the distance is even too short to decelerate from the (possibly smaller than the maximum) current speed to zero. For the movement algorithm to work, an explicit inversion of the movement direction is made, even when the new goal position lies in the direction of movement.

MTN Playback

As soon as an MTN playback command comes in, JointMover leaves the state "Accepting Commands" and goes to the state "Prepare MTN" (see figure 4.3). In this state, Aibo moves to the position specified by the first key frame in the file. The same is done with the last key frame in cases it was not exactly reached after the last execution loop of the MTN file (state "Finish MTN"). Between these two movements (that use the algorithm for manual commands described above) the MTN file is executed as many times as loops are desired (state "MTN"). This is done using Sony's algorithm provided with the `MoNet` sample (see [Samples]). When Aibo has successfully reached the final posture, JointMover goes again to state "Accepting Commands".

As for the manually controlled movements, there is a command vector which is filled with position values and passed to Aibo as soon as the robot is ready. The ready method will cause any further invocations of the MTN movement algorithm.

The MTN file is loaded once from the Memory Stick. Sony provides structures to access the data and a wrapper class with several access methods and a utility method for command vector construction. This method is called every time the MTN movement algorithm is invoked. The method linearly interpolates the positions given for the key frames to obtain the values to be put into the command vector. With a key frame rate of 16 ms, one additional frame has to be put between two key frames. The maximum number of frames (not key frames) that shall be calculated per vector is given as a parameter to the method. The number of key frames taken into account for one command vector is thus this parameter divided by two. The value received in the flag field of the MTN playback command is used as this parameter. This explains why the flag should be at most equal to '16'. The value should at least be '2' to guarantee that one key frame per vector is taken into account. Values not between these two limits are adapted. Uneven values makes the number of frames vary by one between vectors. At the end of the MTN file, a command vector might contain less than the maximum number of allowed frames.

The interpolation method keeps track of the current key frame. The current key frame index is increased after the command vector construction according to the number of frames interpolated. Before each invocation of the interpolation method, the position values for the current key frame (initially zero) given in the MTN file are extracted and the real joint positions are measured. By doing so, it is guaranteed that the measures are made at the very instant the robot is supposed to be in the position given by the key frame index. The MTN algorithm is only invoked when Aibo is ready to accept commands, i.e. when he is not currently executing a command vector and the measured position can't change anymore. The values from the file and those measured are passed together with the current MTN key frame index ('-1' is explicitly passed after the last loop) to RCServer (SetValue entry point on figure 4.2). This explains why the value in the flag field of the playback command also determines the measure rate of the commanded and real joint positions.

4.3 Possible extensions

The modifications proposed in the chapter about the communication protocol would certainly impose modifications to all objects and would possibly need additional objects. An efficient mechanism to handle concurrent access to files would become a necessity. The functionalities offered by Aibo's file system are, however, relatively limited, e.g. locking of files is not possible. For the moment, file access issues have to be considered when giving commands, i.e. an MTN file should not be deleted by RCServer before JointMover has started its execution (deletion after the initial loading is possible).

In the current implementation, JointMover does not give any error feedback to RCServer. Most errors are tracked down by RCServer. For general applications, it is not possible to check every possible error cause before delegating a command to another object. A solution might be to ignore the command in the executing object, as it is done with MTN files having the wrong content. Moreover, RCServer does not check if JointMover is ready when a command coming in over the network is delegated (the check is done for automatically generated individual commands due to a multiple joint command). JointMover treats the received messages much faster than commands can be given over the network. In most general cases, it would be desirable to inform the delegating object of the command outcome before it sends another command, possibly while this object is continuing its own task. For applications including a large amount of objects, orchestration (including file access) becomes a challenge when the concurrency aspect of objects should be kept.

4.4 Contributions to OPEN-R

At the starting date of the project, version 20020201-E-003 of [Model] was the most recent one. While testing sensor reading, it was discovered that the maximum value returned by the two touch sensors on Aibo's head (Head sensor (back) and Head sensor (front)) was 0.980665N, contrary to the value of 2.941995N stated in [Model] (the read maximum value was exactly one third of the documented one). After a posting on the [BBS], Sony corrected the problem by releasing version 20040106-E-004 of [Model], where the actually measurable value replaced the old value.

Following the documentation and the sample programs, it was not possible to give commands to the two tail LEDs of the Aibo robot used for the projet. A message was posted on [BBS] but Sony could not provide any useful information yet. It is not clear whether this is a problem specific to the used robot or a general OPEN-R bug.

Chapter 5

Aibo Remote Control

5.1 wxWindows

wxWindows is a C++ framework providing GUI (Graphical User Interface) and other facilities on more than one platform. Version 2 currently supports all desktop versions of Windows, Unix with GTK+, Unix with Motif, and Mac OS. Visit the web page [wxWin] to learn more.

wxWindows was originally developed at the Artificial Intelligence Applications Institute, University of Edinburgh, for internal use, and was first made publicly available in 1992. Version 2 is a vastly improved version. According to the developers, wxWindows was developed to provide a cheap and flexible way to maximize investment in GUI application development and meets all of the following criteria:

- low price (free)
- source availability
- simplicity of programming
- support for a wide range of compilers

As open source software, wxWindows has benefited from comments, ideas, bug fixes and enhancements of users. This gives wxWindows a certain advantage over its commercial competitors, plus a robustness against the transience of one individual or company. This openness and availability of source code is especially important when the future of a developed application may depend upon the longevity of the underlying class library.

The importance of using a platform-independent class library cannot be overstated, since GUI application development is very time-consuming, and sustained popularity of particular GUIs cannot be guaranteed. Code can very quickly become obsolete if it addresses the wrong platform or audience. Although wxWindows may not be suitable for every application, it provides access to most of the functionality a GUI program normally requires, plus many extras such as network programming, PostScript output, and HTML rendering. Some other features of wxWindows are:

- simple-to-use, object-oriented API

- flexible event system
- constraint-based and sizer-based layouts
- toolbar, notebook, tree control, advanced list control classes
- common dialogs for file browsing, printing, colour selection, etc.
- support for many file formats (e.g. BMP, PNG, JPEG, GIF, XPM)
- over 50 sample programs
- over 1000 pages of printable and online documentation

5.2 Client software

The part of the remote control system that turns on the PC is an application called "Aibo Remote Control" (the executable file is called `AiboApp`). It has two main components, one that handles all the network traffic with Aibo (with the RCServer object to be precise) and one that enables the graphic display of all interface elements. Figure 5.1 makes this differentiation clear.

5.2.1 NetworkConnector

The interface accesses networking functionalities by method invocation on the `NetworkConnector`. The method is called in the event handler of the interface element that triggers the desired command. In most situations, the parameters passed in the methods correspond to the content of the message to be sent. There is a different method for each message type. Additional methods for connection establishment and closure exist. Parts of the code are based on the client part of Sony's `EchoServer` sample program in [Samples]. Sony, however, did not define any timeouts. In the current implementation, the NetworkConnector has a two seconds timeout on send and receive operations.

The interface is blocked until the command is processed and the answer is received from Aibo. This synchronized communication scheme between Aibo and the client guarantees that another command can only be sent to Aibo when the previous response was received and Aibo is thus ready to treat the new command.

NetworkConnector checks the outcome of the command and informs the interface when there was an error. An error message is then displayed in the interface and appropriate measures are taken, in most cases connection closure. Otherwise the out parameters of the called method are used to update the display according to the received value. On big-endian machines, NetworkConnector automatically does any necessary transformations from little-endian.

5.2.2 AiboFrame

For the interface design, various classes have been created. Figure 5.1 also shows, what classes are used in the interface (`AiboFrame`) and how they are used to build other classes. One inheritance relation is shown (case of the Aibo pictures). The figure does not represent a complete concept diagram and does

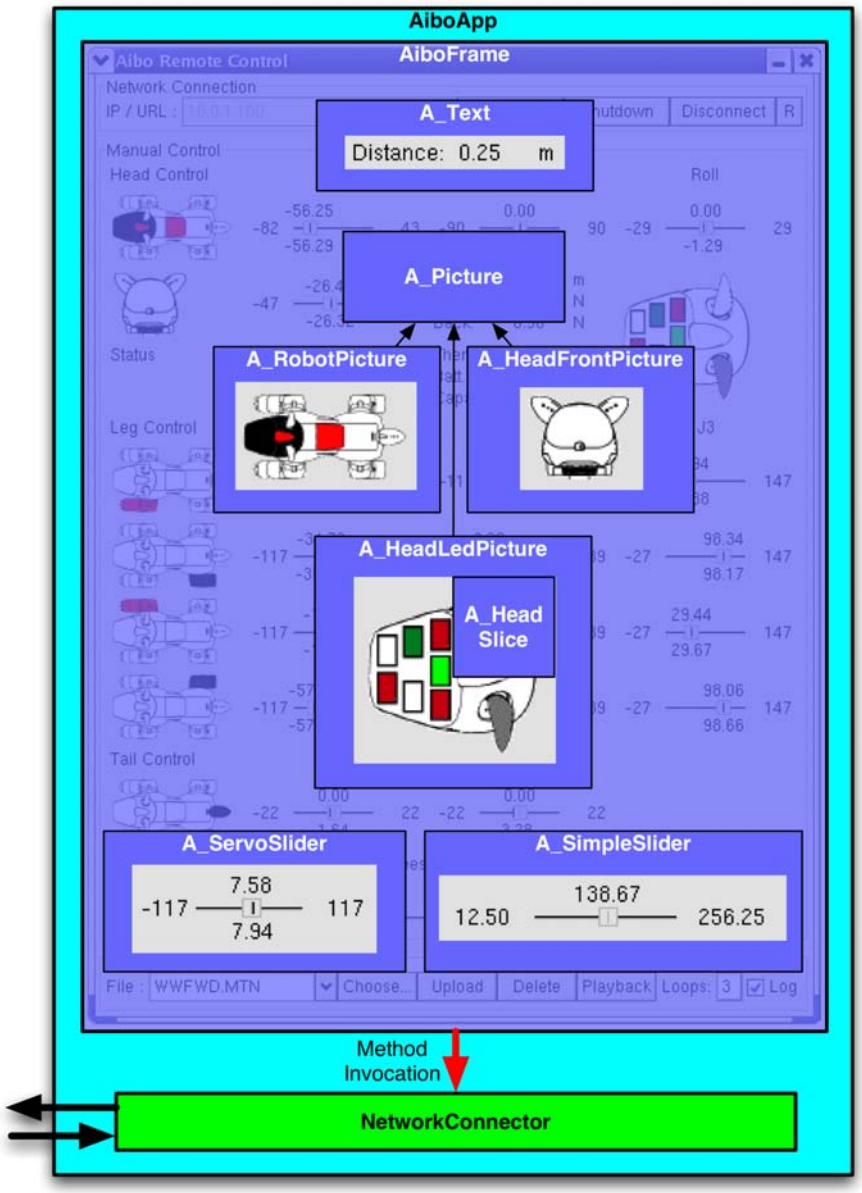

Figure 5.1: Structure of Aibo Remote Control

only show what classes are used for. In the implementation, AiboFrame has a reference of all the objects in the interface that are directly accessed. In some cases (sliders, LEDs) a reference of AiboFrame is stored in the classes for doing callback invocations.

AiboApp contains one single instance of AiboFrame. (Static)BoxSizers and GridSizers are used to arrange the interface elements. Figure 5.2 shows the initial state of all controls. When the client software is not connected to Aibo, most of the controls are disabled. On figure 5.3 it can be seen how the interface looks like after some interaction.

The interface is divided into three parts with logically grouped controls. The following paragraphs follow approximatively this division. They describe how the functionalities of RCServer and JointMover are accessed.

Networking, Shutdown and Reboot

The default IP number is already entered in the address field upon startup of the application. The button "Connect" establishes a connection via Network-Connector. All the controls become active and the button changes its name to "Disconnect" when the operation is successful. Remember that LEDs an plungers are set to their initial values by Aibo when a connection is established. The interface reflects this by resetting them to these values upon a click on the "Connect" button.

"Shutdown" and "Reboot" trigger the corresponding action on Aibo. The robot still can send the answer corresponding to the command before he closes the network connection. The connection is also closed on the client upon reception of the answer message. As Aibo will go to its initial posture after reboot (due to the command or manually), all the joint controls are reset to reflect this initial position. If ever it is necessary to reset the controls without relaunching the application (i.e. Aibo unexpectedly closed the connection), the button "R" can be used for this purpose.

Sensors

When a connection is opened to Aibo, observation of all sensors is automatically turned on. No interface for reading single values and changing the observation state of individual sensors is included in Aibo Remote Control. In addition to the starting of the observation, all the current acceleration and velocity limits are requested (individually) from Aibo and the maximum value of each kind of limit is used as the starting position of the two corresponding sliders. This ensures that these values are correctly displayed even when the application was relaunched and reconnected to an Aibo whose limits were previously modified.

As already stated, Aibo does only send messages when they are explicitly requested. This means that continuous sensor reading requires that the message `s 0` is repeatedly sent to Aibo, possibly interleaved with command messages. A simple solution was chosen to guarantee this. wxWindow defines an idle event. This event occurs every time the interface is idle, i.e. no interaction events are produced. The reading of all sensors is done in the handler of the idle event. The synchronized communication scheme between Aibo and the client ensures that the client can't congest Aibo with requests. Not having a separate thread for sensor query means that no concurrency issues must be considered on the

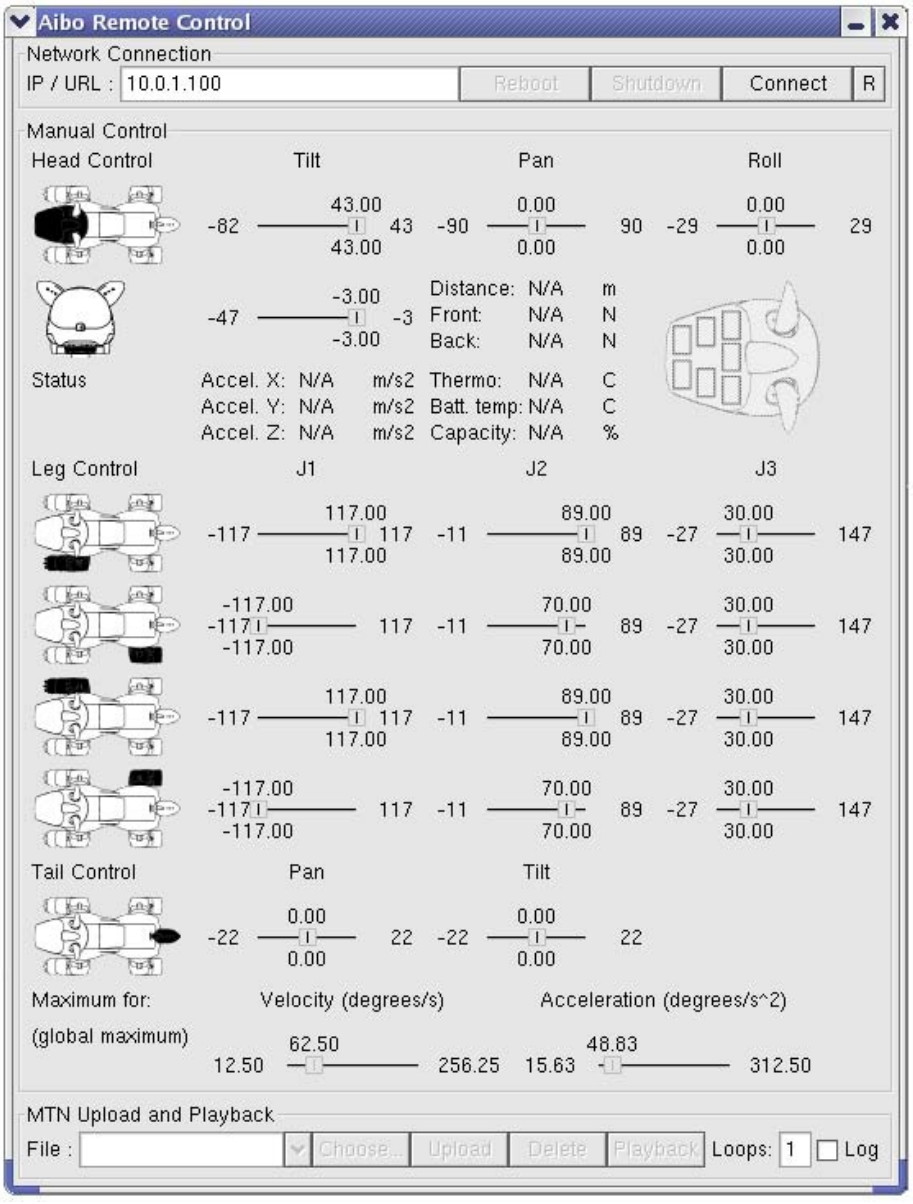

Figure 5.2: ScreenShot Aibo Remote Control (disconnected)

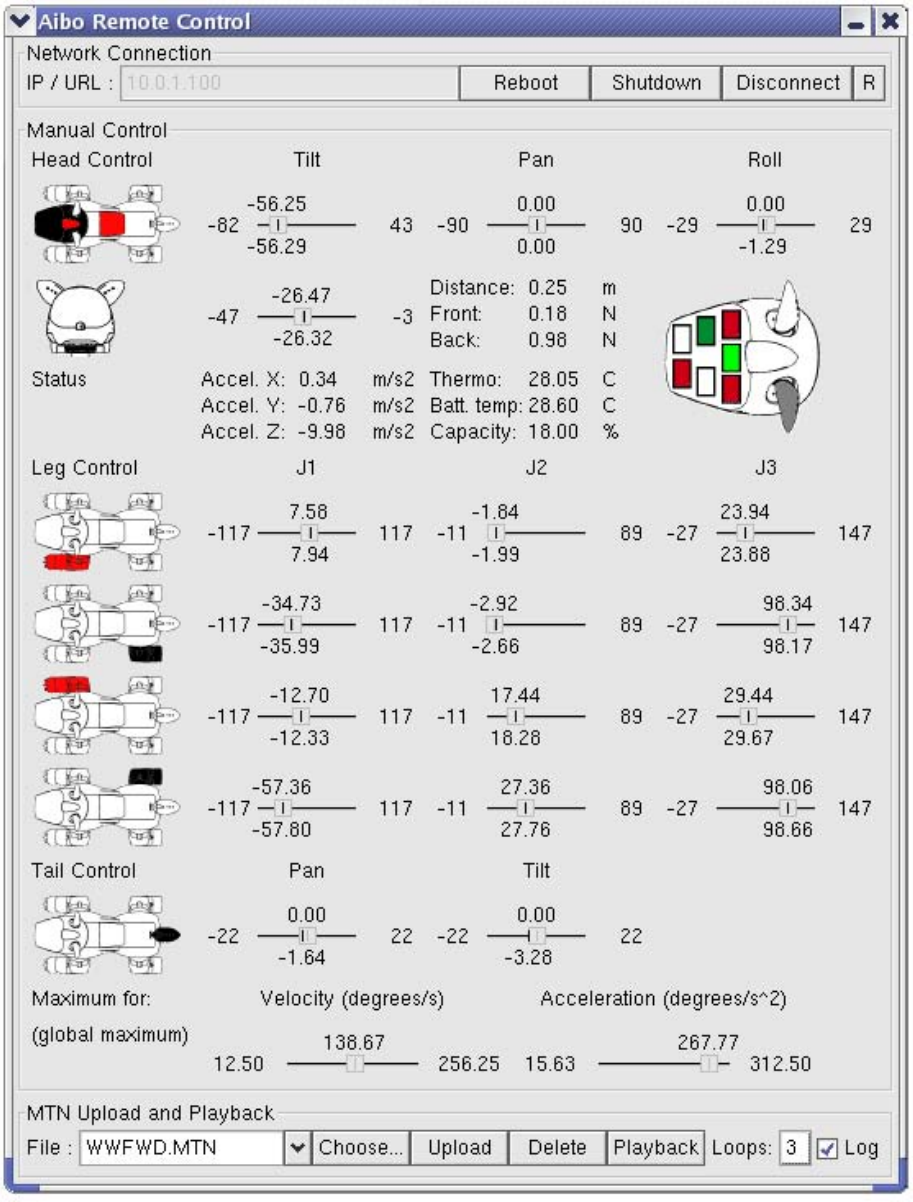

Figure 5.3: ScreenShot Aibo Remote Control

client. The drawback of the idle event approach is that the update frequency is highly dependent on the performance of the client machine.

The result of giving the command to read all sensors is an array of shorts (see table 3.10). Before the values are displayed at the correct location, they are of course transformed to have the correct unit, i.e. a division by 100 has to be done in most cases.

The joint values are used to set the position indicators of the joint sliders. These are the vertical lines (and associated numerical values) that lie most of the time under the manually moveable part of the slider. The moveable part represents the commanded goal position. Especially with low velocities and accelerations, it can be observed, how the position indicator slowly reaches the goal position.

Every binary sensor is associated with one of the robot pictures. The pictures are composed of different slices whose appearances can be changed individually. In general, there are three different possible bitmaps per slice, one in neutral state, one in selected state and one in active state. The neutral state is just used for illustration purposes. The selected state is used to specify a certain robot component, e.g. the leg controlled by a row of sliders is filled with black color in the picture in front of the row. When the binary sensor corresponding to the component is activated, the selected slice becomes active, e.g. when a paw sensor is activated, the corresponding leg changes its color from black to red. The two top pictures are used to display the states of the chin and back switches.

Non-binary sensors (including remaining battery capacity and battery temperature) each have a textual display. The two touch sensors on top of Aibo's head are in plus linked to the picture which is also used for the display of the back switch. When either of the two head sensors reports a non-zero value, the sensory region is displayed in red color.

LEDs and Plungers

Head lights and ears are represented almost like the other robot bitmaps, but in addition the slices process click events. For this reason a class representing these slices had to be introduced (see figure 5.1). The LEDs are represented by rectangles on a detailed picture of Aibo's head. The ears have their usual form. A slice and thus the clickable area is bigger than the graphical representation of the controlled element. The slices change their appearance when they are clicked. Depending on the previous state, they go to state on or off and a corresponding individual LED or plunger command is sent to Aibo (by a method in AiboFrame which is called back by the slice).

Joints

Every joint has an associated slider. These sliders contain a position indicator which is set according to the read sensor value. The moveable part of the slider (and the associated numerical value) indicates the desired goal position Aibo will go to with respect to velocity and acceleration limits. The slider processes various mouse events and calls back a method in AiboFrame that will cause an individual joint command to be sent.

At both ends of the slider's range the position limits are indicated. These are rounded values to save space. The exact value will be displayed next to the moveable part when the slider is put to the limit.

There is a slider for the global maximum velocity limit and one for the acceleration limit. When a new value is set, corresponding commands to set the limits for all joints are sent to Aibo. As described in previous chapters, the individual limits will be respected, the displayed value is the maximum over all limits.

As already stated, upon connection to Aibo, all the current acceleration and velocity limits are requested (individually) and the maximum value is used to set the position of the corresponding sliders. Nothing similar is done for the position sliders, because their position indicator already reflects the measured sensory value. As a consequence, after an application relaunch, there might be a large difference between the measured position and the current position of the moveable slider part, because the latter indicates the initial position.

MTN files

The bottom part of AiboFrame allows the upload and playback of MTN files. Only files with an `*.MTN` name extension can be selected in the dialog opened by the "Choose..." button. The dialog proposes the directory "mtn" in the application directory as the starting point. The chosen files are added to a menu.

A file chosen from the menu can be uploaded with the corresponding button. Depending on the file size, the interface might block for a moment. To remove a file from the menu (and from Aibo's Memory Stick if the file exists there, i.e. after a successful upload), the "Delete" button is used. All the necessary parameters (complete file path, file size for upload) are handled internally and the correct values (filename, number of packets) are written to the fields of a file command message.

When the "Playback" button is pushed, some interface elements are disabled for the duration of the file execution, e.g. the other file handling buttons, the joint sliders and the button for disconnection. The observation of the MTN key frame is automatically turned on by sending the appropriate command. Therefore more values are received when the sensors are read. Sensory display works as before, with the difference that the commanded joint positions (which are now received in addition) are used to set the positions of the normally manually moveable joint slider part (manual control is disabled for the duration of the playback).

The playback command takes into account the numeric value in the "Loops" field. Only numeric values can be entered and there is an error message when the value surpasses the capacity of the field type used for transmission (the value can not be greater than 65535). To stop Aibo before the last loop is done, a second playback command with the loops value equal to zero must be given.

As soon as the MTN key value is again equal to '-1' (the last loop was executed), the observation of the key frame is turned off and the disabled interface elements are again activated. The '-1' values received while Aibo was moving to the first key frame position were ignored.

5.2.3 Logfile

The checkbox "Log" activates the recording of some of the received values during MTN execution, i.e. values are only recorded when the MTN key frame is valid. It is possible to receive twice the same key frame, therefore only new frames are recorded. The key frame rate is determined by the value flag of the playback command. This value has been fixed to '8', i.e. the key frame rate is 4.

In addition to displaying the sensor data and the commanded values, they are appended to the file `LOG.LOG` in the application directory. The values are written in little-endian format and in the units given by table 3.6, i.e. the values are used as they are received from Aibo. The recorded values are (in this order): the MTN key frame, the commanded position values for the 15 MTN joints, the measured position values for the 15 MTN joints, the values of the four paw sensors. The order of the joints is given by table 3.3, those of the paw sensors by table 3.4. At every key frame, 35 values of type short are written consecutively to the file. Only a header of 8 characters is written at the beginning of the file: `AIBO_1.0`. This allows distinction from possible future file formats recording other values.

5.3 Possible extensions

As soon as it is known how to command tail LEDs, corresponding interface elements should be added. The existing elements could be graphically adapted to better reflect their disabled state.

The current communication protocol does not need multiple threads. Introduction of dedicated threads would provide more detailed control than usage of the idle event. This would enable more powerful networking possibilities, specially once multiple connections had to be handled in parallel. With a supplementary connection for continuous unrequested sensor data reception, a fundamental redesign of the client's networking facilities would be inevitable. As a drawback of having multiple threads, concurrent access problems in the interface must be handled in such cases.

Bibliography

[Start] "Aibo ERS-210: QuickStart Manual" (see section A.2)

[OPEN-R] Official Sony OPEN-R web page:
`http://openr.aibo.com/`
Direct link to the English web page:
`http://openr.aibo.com/openr/eng/index.php4`

[BBS] Section "Bulletin Board" on [OPEN-R]

[Download] Section "Download" on [OPEN-R]

[FAQ] Section "Frequently Asked Questions (FAQ)" on [OPEN-R]

[Notes] Section "Important notes when using the OPEN-R SDK" on [OPEN-R]

[Docs] File "OPEN-R SDK Documents English":
`OPEN_R_SDK-docE-XXX.tar.gz`[1] in [Download]

[Install] Document "Installation Guide":
`InstallationGuide_E.pdf` in [Docs]

[Prog] Document "Programmer's Guide":
`ProgrammersGuide_E.pdf` in [Docs]

[Model] Document "Model Information for ERS-210":
`ModelInformation_210_E.pdf` in [Docs]

[Ref] Document "Level2 Reference Guide":
`Level2ReferenceGuide_E.pdf` in [Docs]

[IP] Document "OPEN-R Internet Protocol Version4":
`InternetProtocolVersion4_E.pdf` in [Docs]

[Revision] Document "Revision Record":
`RevisionRecord_E.pdf` in [Docs]

[Samples] File "Sample Programs":
`OPEN_R_SDK-sample-XXX.tar.gz` in [Download]

[wxWin] Official wxWindows web page:
`http://www.wxwindows.org/`

[1] "XXX" in a filename stands for the current version number of the file (not the same for all files)

Appendix A

Additional information

A.1 MTN file format

The file format as it is show here is defined in the file `MTN-FFORM-E.txt` coming with the `MoNet` sample program included in [Samples].

```
- function: Provides motion data information for a robot.

- byte order: little endian

- data type definition:

  type: Strings are composed by the following definition.

    offset  size    type    contents                        data
    ------+-------+-------+-------------------------------+----
       0      1     uc      length
       1      x     ch      characters

    length: The number of characters in a string
    characters: String data

- File Format:

    offset  size    type    contents                        data
    ------+-------+-------+-------------------------------+----
       0      1     ch      magic 0                         'O'
       1      1     ch      magic 1                         'M'
       2      1     ch      magic 2                         'T'
       3      1     ch      magic 3                         'N'
       4      24    cp      Section 0
       28     x     cp      Section 1
       x      x     cp      Section 2
       x      x     cp      Section 3

    magic 0-3: file magic
```

1. Section 0
Section0 is the header of fundamental items.
It has a fixed length.

offset	size	type	contents	data
0	4	lw	Section Number	0
4	4	lw	Section Size	24
8	4	lw	Number of Sections	4
12	2	wd	Major Version	1
14	2	wd	Minor Version	2
16	2	wd	Number of keyframes	
18	2	wd	Frame Rate	16
20	4	lw	Reserved	0

Section Number: Serial Number of this section
Section Size: The size of this section (in bytes)
Number of Sections: The number of sections in this file
Major Version: The major version number of Format
Minor Version: The minor version number of Format
Number of keyframes: The number of keyframes in this file
Frame Rate: Motion replay speed [msec/frame]

2. Section 1
This header describes the fundamental names.
It has a variable length.
Note: Use padding when the Section Size is less than 4 bytes.

offset	size	type	contents	data
0	4	lw	Section Number	1
4	4	lw	Section Size	
8	x	Strings	Motion Name	
x	x	Strings	Creator	
x	x	Strings	Design Label	
x	0-3		Padding	0

Section Number: Serial Number of this section
Section Size: The size of this section (in bytes)
Motion Name: file name without the mtn extension
Creator: Author name
Design Label: The design label of the robot

3. Section 2
This header describes the Joint list.
It has a variable length.

Note: Use padding when the Section Size is less than 4 bytes.

offset	size	type	contents	data
0	4	lw	Section Number	2
4	4	lw	Section Size	
8	2	wd	Number of Joints	
x	x	Strings	PRM 0	
x	x	Strings	PRM 1	
x	x	....		
x	x	Strings	PRM (Number of Joints -1)	
x	0-3		Padding	0

Section Number: Serial Number of this section
Section Size: The size of this section (in bytes)
Number of Joints: The number of Joints used in this file
PRM: CPC Primitive Locator name of a Joint used in this file

4. Section 3
This head describes the type of motion data.
It has a variable length.
Note: Use padding when the section Size is less than 4 bytes.

offset	size	type	contents	data
0	4	lw	Section Number	3
4	4	lw	Section Size	
8	4	lw	Data Type	0
12	x	cp	Data Section	

Section Number: Serial Number of this section
Section Size: The size of this section (in bytes)
Data Type: The kind of Motion Data (0 : angle line data)

4-1. Data Section
The following is the data format when the Data Type is 0.

```
[Roll] [Pitch] [Yaw]
[Joint Data]
{
  [Number of Interpolate frame]
  [Roll] [Pitch] [Yaw]
  [Joint Data]
}  X (Number of Keyframes - 1)
```

offset	size	type	contents	data
0	4	lw	Roll	
4	4	lw	Pitch	
8	4	lw	Yaw	

```
  12      x       cp      Joint Data Section
  x       4       lw      Number of Interpolation frames
                            .....
  Roll:  Robot's BODY roll (in Micro radian)
    (when it leans to the right, plus)
  Pitch: Robot's BODY pitch (in Micro radian)
    (when it leans backward, plus)
  Yaw:  Robot's BODY yaw (in Micro radian)
    (when it turns to the left, plus)
  Joint Data Section:
    The joint data is associated with the corresponding PRM
    String in Section 2.
  Number of Interpolation frames:
    The number of interpolation frames with the former frame.

4-1-1. Joint Data Section

  offset  size    type    contents                          data
  -------+-------+-------+---------------------------------+----
  0       4       lw      Joint Data 0
  4       4       lw      Joint Data 1
                          .....
  x       4       lw      Joint Data (Number of Joints)

  Joint data: The position of a joint (in Micro radians)
```

A.2 List of added documents

1. "Aibo ERS-210: QuickStart Manual"