The URBI Tutorial

v 1.0

October 2005

Contents

Acknowledgment

I would like to thank Matthieu Nottale for his contribution to the liburbi-C++, URBILab and so many other things, Bastien Saltel for liburbi-Java and liburbi-OPENR, Anthony Truchet for the URBI4Webots adaptation of URBI, Martin Cacky for the early work on URBILab, François Serra who first helped in deciphering the OPENR sanctorum, Sony CSL, LRV, Aldebaran Robotics and all contributing research labs and hobbyists for their feedback and encouragement.

Jean-Christophe Baillie
Oct. 2005

The URBI Team,
www.urbiforge.com

Introduction

URBI (Universal Robotic Body Interface) is a scripted interface language designed to work over a client/server architecture in order to remotely control a robot or, in a broader definition, any kind of device that has actuators and sensors. As it will be shown in this tutorial, URBI is more that a simple driver for the robot, it is a universal way to control the robot, add functionalities by plugging components and develop a fully interactive and complex robotic application in a portable way.

The main distinctive qualities of URBI are the following:

- **Simplicity**: easy to understand, but with high level capabilities
- **Flexibility**: independent of the robot, OS, platform, interfaced with many languages (C++, Java, Matlab...)
- **Modularity**: soft devices are available to extend the language
- **Parallelism**: Parallel processing of commands, concurrent variable access policies, event based programming,...

Probably one of the most important point for this tutorial is the first one: URBI has been designed from the beginning with a constant care for simplicity. There is no "philosophy" or "complex architecture" to be familiar with. It is understandable in a few minutes and can be used immediately.

URBI is available with many robots and the number is increasing. Currently, there is an URBI version for Aibo, for the HRP-2 humanoid robot, for the Webots universal simulator and the Pioneer robots and other humanoids are on the way.

The Webots simulator compatibility means that it is possible to switch from the real robot to simulation with a simple IP address change, and this makes URBI particularly suitable in that case. See `http://www.cyberbotics.com`.

In this tutorial, we have tried to make a step by step description of URBI which goes from simple motor commands up to more complex programming and software components integrated in URBI. It is meant to be understandable by people having little or no background in robotics and programming (except for the C++ sections, which require that you understand C++ at a basic level). However, from time to time, we have inserted explanations or complements that will probably make sense only for advanced users or academics/industrials. These inserts are presented with a small academic sign as shown on the left of this text.

Chapter 1

Installing URBI

We cannot detail in this tutorial how to install URBI for any particular robot type, but the general idea is to have the URBI server program loaded and running on your robot and the process to do so should be described in the INSTALL file of the package you have downloaded. In the ideal situation, URBI is preinstalled on your robot anyway.

Since we will use many Aibo examples in the tutorial, we give here the instructions on how to install URBI on an Aibo robot. We also describe how to install URBILab which is a simple and convenient cross-platform graphical client to replace telnet.

1.1 Installing the memorystick for Aibo

First, download the precompiled memorystick for your specific robot. There are two possibilities at the moment:

- ERS2xx : `http://www.urbiforge.com/ers200`
- ERS7 : `http://www.urbiforge.com/ers7`

Quick instructions:

unzip the archive and put the content of the MS-xxx directory on a blank memorystick, updating the `WLANCONF.TXT` file with your specific network config.

Detailed instructions:

1. Untar/Unzip the memorystick archive corresponding to your Aibo. You should get a directory named `MS-ERS7` or `MS-ERS200`. Enter into this directory.

2. From the `MS-ERS7` (or `MS-ERS200`) directory, go to the `OPEN-R/SYSTEM/CONF` directory. There should be a `WLANCONF.TXT` file here (or you must create it), to configure the network properly. There is no official documentation on the how to write the `WLANCONF.TXT` file, but here is an example that you can customize for your robot:

```
HOSTNAME=aibo.mydomain.com
ETHER_IP=192.168.1.111 # <-- your IP here
#
# WLAN
#
ESSID=0a3902 # <-- your SSID here
WEPENABLE=1 # <-- WEP or not
WEPKEY=0x4B2241785B # <-- the key:  hexa
#WEPKEY=ABCDE # <-- ASCII with ERS2xx
APMODE=1

#
# IP network
#
USE_DHCP=0
SSDP_ENABLE=1

# This part can be omitted
# Your network config here -->
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.0.3
DNS_SERVER_1=192.168.1.1
```

   You can use URBI on Aibo without the network if you don't have a wifi access point, by putting your URBI programs in the `URBI.INI` file.

3. Copy the content of the `MS-ERS7` or `MS-ERS200` directory in the root of a blank programmable pink memorystick (a "PMS")[1]. Be careful that this is *not* the Aibo Mind memorystick or one of the blue memorysticks: actually, you must go and buy a specific aibo programming memorystick from Sony, it is unfortunately not included in the Aibo package. Then, put this memorystick in the robot and start it. Your URBI robot is ready.

   You can run `telnet`[2] on port 54000 of the robot to check if everything is OK:

[1] MS-ERS7 or MS-ERS200 should not appear on the memorystick, only the content of these directories must be copied in the root of the memorystick

[2] Note for windows users: the command line `telnet` in windows doesn't work very well and you should use URBILab, described in next section, instead of `telnet`. However, `telnet` works fine with MacOSX or linux.

```
telnet aibo.mydomain.com 54000
```

You should get a URBI Header at start, which looks like this:

```
[00139464:start] ************************************************************
[00139464:start]  URBI Language specif xxx &     Copyright (C) 2005 JC Baillie
[00139464:start]  URBI Kernel version yyy
[00139464:start]
[00139464:start]    URBI Server version zzz for Aibo ERS2xx/ERS7 Robots
[00139464:start]    (C) 2004-2005 Jean-Christophe Baillie
[00139464:start]
[00139464:start]  URBI comes with ABSOLUTELY NO WARRANTY;
[00139464:start]  This software is free, and you are welcome to use
[00139464:start]  it under certain conditions; see LICENSE for details.
[00139464:start]
[00139464:start]  See http://www.urbiforge.com for news and updates.
[00139464:start] ************************************************************
[00139464:ident] ID: U597766392
```

1.2 Installing URBILab

One interesting benefit of the client/server architecture of URBI is that you can start right away to send commands to your robot with a simple `telnet` client. It is of course possible to interface URBI with a C++ or Java program, which will be described later with the liburbi (chapter 6), but for the moment we will use a simple `telnet` interface.

However, `telnet` is a very crude and limited client (which does not always work well under windows[3]), and we have developed a cross-platform graphical alternative called **URBILab** that you are encouraged to use. URBILab is free and released under a GNU-GPL License. You can download it here:

`http://www.urbiforge.com/urbilab`

The URBILab application is divided into three panes: a message pane which contains messages from the URBI server, a command pane where the commands you have sent are visible and a command line where you actually type commands in. The command line has history that you can access with the arrows, like a unix shell. Fig. 1.1 shows a screenshot of URBILab.

[3]It works if you use cygwin, otherwise carriage returns are badly interpreted by the native windows implementation of telnet

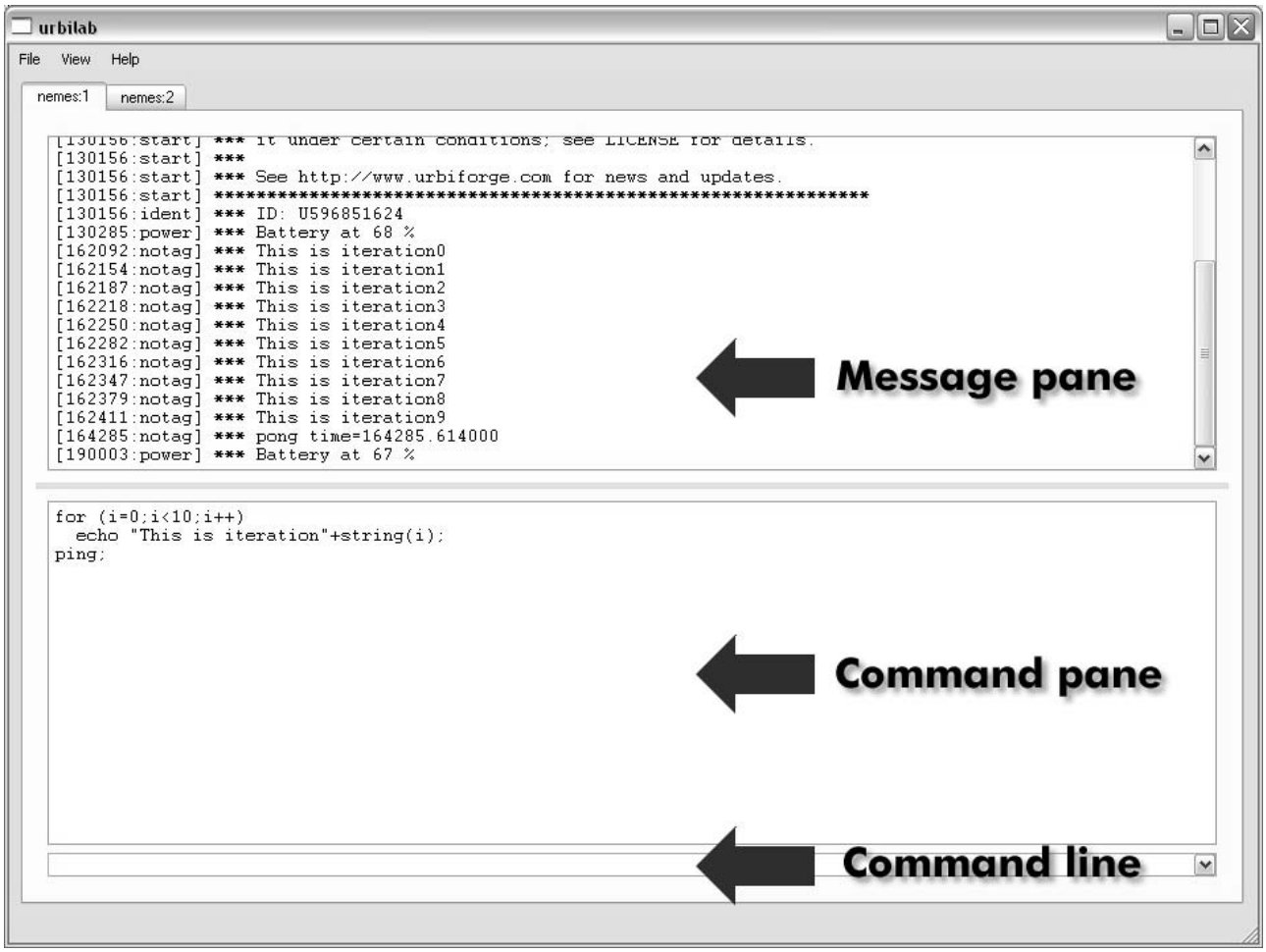

Figure 1.1: URBILab

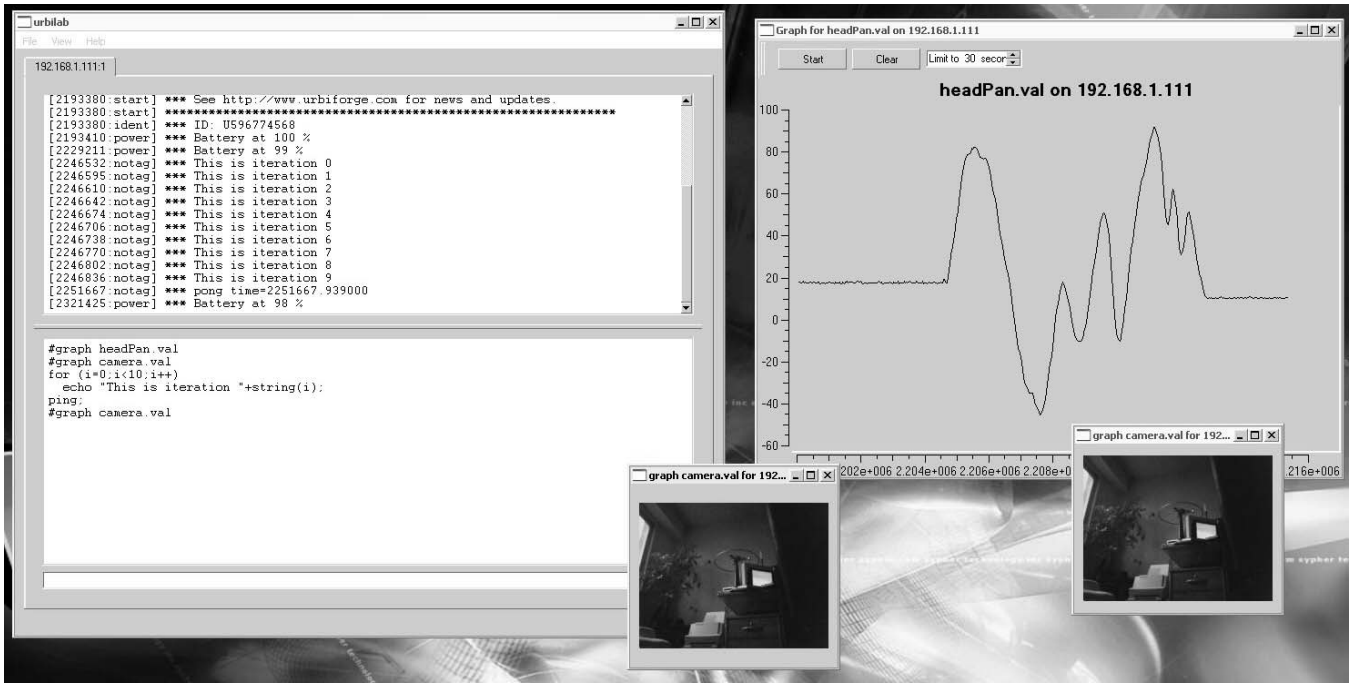

Figure 1.2: Numerical variables or image variables graphing

Each time we will show URBI commands in the rest of this tutorial, you can type those commands in the command line pane of URBILab to try them (or in a simple `telnet` session), there is nothing more needed to start using URBI.

One interesting feature of URBILab, beyond being a graphical cross-platform telnet, is that it allows to graph variables with the `#graph` command prefix. You can graph `camera.val` which will open a window with a video feedback from the robot at 30fps. You can graph a numerical variable, like a joint value `headPan.val`, and a window appears with a graph of this value over time:

```
#graph camera.val
#graph headPan.val
```

Of course, these `#graph` commands are URBILab specific commands and are not part of the URBI language. Fig. 1.2 illustrates the graphing possibilities.

Chapter 2

First moves

In the following, we will use examples from the Aibo robot, but you can easily transpose them to your particular robot. Each element of the robot (sensors, motors, camera,...) is a device and it has a device name. In the Aibo, you have devices for the head motors called `headPan` and `headTilt`. The camera device is called `camera`.

You can find out what devices are available for your particular robot by checking the associated *URBI Doc* file, or simply by typing the command `devices;`. Also, the command `info device;` will display information about the given device.

2.1 Setting and reading a motor value

We will make use of the motors in the following, so first of all we have to start them:

```
motor on;
```

Let's start by moving the `headPan` motor to 30 degrees:

```
headPan = 30;
```

Now, let's ask what is the value of the `headPan` device:

```
headPan;
[139464:notag] 30.102466
```

The server responds with a *server message* (written in italic font here to make it easier to distinguish it from commands) prefixed by a timestamp and a tag between brackets. Since there is no tag associated to the command in this example, `notag` is used by default. It is very simple to associate a tag to a command in URBI by prefixing the command with the tag and a colon:

```
mytag:headPan;
[139464:mytag] 30.102466
```

The message has now the `mytag` tag. This will be crucial to know who is sending what when several commands are running in parallel.

You can try to set different motors, like `legRF1` or `tailTilt`, or play with LEDs like `ledF1` or `ledBMC`, or read sensor values like the distance detector `distanceNear` or the accelerometer `accelX`, `accelY`, `accelZ`. The syntax is always the same: `device = value;`.

2.2 Setting speed, time or sinusoidal movements

The above examples set the value of the device as fast as the hardware of the robot allows it. You might want to do more complicated things like reaching a value in a given time (in milliseconds):

```
headPan = 30 time:3000;
```

Which will reach the value 30 degrees in 3000ms. Whenever you need to express a time value in URBI, you can explicitly use units like this:

```
headPan = 30 time:3s;
headPan = 30 time:3000ms;
headPan = 30 time:3m;
headPan = 30 time:3h26m15s;
```

You can compound at will days (d), hours (h), minutes (m), seconds (s) and milliseconds (ms), with decimal values. By default the unit is milliseconds if no unit is specified or when a variable expression is used.

Alternatively, you can set the speed used to reach the value, expressed in $units/s$:

```
headPan = 30 speed:1.4;
```

Or the acceleration (expressed in $units/s^2$):

```
headPan = 30 accel:0.4;
```

One very useful way of assigning a variable is with a sinusoidal oscillation:

```
headPan = 30 sin:2s ampli:20,
```

This will make the `headPan` device oscillate around 30 degrees with an amplitude of 20 degrees and a period of 2s. Note that the command ends with a comma and not a semicolon. We will explain why after, but the reason is that the sinusoidal assignment never terminates and the comma sort of "puts it in background" to allow other commands coming after it to be executed. Otherwise, with a semicolon, nothing coming after this sinusoidal assignment could be executed.

`time`, `speed` or `sin` are called modifiers. Many other modifiers are available like `phase`, `getphase` or `smooth`. Check the *URBI Language Specification* for a completed description of modifiers.

One particularly powerful modifier is `function` which assigns an arbitrarily complex function of time as the variable trajectory. This is described in the *URBI Language Specification* and will only be available in servers with kernel 1.0 or above

2.3 Discovering variables

You can use variables in URBI. Simply assigning a value to `x` will create a variable `x` local to your connection:

```
x = 4;
x;
[146711:notag] 4.000000
```

General structure for variables

Variable names are always of the form `prefix.suffix` and when no prefix is supplied, a prefix local to the connection is silently added so that `x` in one connection will not interfere with `x` in another connection.

For example, when you type `x` URBI will in fact use `U596851624.x` in its memory, with `U596851624` being the identifier of your current connection (where you typed `x` in). In the same way, function calls have a local namespace attributed, so that you can do recursive function calls without interferences.

Device values and automatic `.val`

There is one important exception to the rule saying that variables without prefixes are local: when you type `headPan`, URBI do not treat this as a local variable, but instead it recognizes that `headPan` is a device name and it transforms the expression into `headPan.val`, which is a standard URBI variable containing the device value. So, in reality, `headPan` do not refers to a local variable but to the global variable `headPan.val`. This applies to any device or virtual device and it is meant to simplify the look and feel of standard URBI code for beginners. You can decide to use `device.val` instead if you want to be explicit about the variable you are using.

Making "global" variables

There is no real concept of local or global variable in URBI, as we have explained. Everything is of the form `prefix.suffix`. Without prefix, the variable is local to the connection but you can use your own prefix to make your variable "global":

```
myprefix.x = "hello";
```

Expressions

Note that the type of the variable (numeric, string, list or even binary as we will see later) is automatically inferred by URBI.

You can evaluate arbitrary complex expressions, including variables or known functions like `sin`, `cos` or `random` (see the URBI Specification for a complete list):

```
x=pi/2;
calc:sqrt(1+sin(x));
[148991:calc] 1.414213
```

One interesting feature is that modifiers in complex assignments are constantly reevaluated so that if they contain variables, the value of the modifier might change over time as the corresponding variable is evolving. Consider the following example which assigns to `x` a sinusoidal oscillation within a sinusoidal envelop between 15 and 25:

```
the_amplitude = 20 sin:10s ampli:5,
x = 0 sin:2s ampli:the_amplitude,
```

Complex interactions between variables and devices value can be established with this capability.

2.4 Lists

You can store several elements in a list with URBI, simply by putting them between brackets:

```
mylist = [1,2,35.12,"hello"];
mylist;
[139464:notag] [1.000000,2.000000,35.120000,"hello"]
```

You can easily add elements or add lists:

```
mylist = [1,2] + "hello";
mylist;
[146711:notag] [1.000000,2.000000,"hello"]
x = 1;
mylist + [45,x];
[148991:notag] [1.000000,2.000000,"hello",45.000000,1.000000]
```

Then, you can scan the content of a list with a `foreach` command:

```
list = [1,2];
foreach n in list { echo n };
[151228:notag] *** 1.000000
[151228:notag] *** 2.000000
```

Note that, for technical reasons, the code executed in the `foreach` command must be enclosed between brackets.

2.5 Running commands in parallel

Commands in URBI can last during a certain amount of time. We have seen so far that we can assign values with a certain time and with a certain speed, or even assign values in a sinusoidal way, which lasts forever. There are many ways in URBI to have these commands running in parallel. We have already seen how to do it by using a comma to separate commands instead of a semicolon.

There is another way to specify that commands should be run in parallel, by using the `&` separator:

```
x=4 time:1s & y=2 speed:0.1;
```

The difference with the comma separator is that `&` forces the two commands to start at exactly the same time. In particular this means that the first command cannot start until the second command is fully available. So, typing `x=4 time:1s &` in the console will not start anything, because URBI is waiting for what comes next, after the `&`.

In the same way commands can be run serially, exactly one after the other, by using a pipe separator:

```
x=4 time:1s | y=2 speed:0.1;
```

There will be no time gap between the two commands so, here again, URBI waits for the second command to be available: unlike the semicolon separated commands, the second command must start exactly after the first, so it must be ready in advance.

Using semicolon or comma separators is more permissive, because it will start immediately any command before the separator. But you might need strong time synchronization constraints, and that's why `&` and `|` separator are here for.

Note that you can group commands between brackets and build a more complex architecture of parallel and serial commands, like this:

```
{ { x=4 time:1s | y=2 speed:0.1 } &
  z = 0 sin:200ms ampli:4 } |
t = 2,
```

TIP: In general, it is a good idea to end commands entered in a console (URBILab or telnet) by a comma, to avoid blocking the connection after entering a never ending command.

2.6 Conflicting assignments

Since it is possible to run commands in parallel, possible conflicts might arise. For example, what will happen if something like this is executed?

```
x=1 & x=5;
```

`x=5` is a conflicting assignment since it accesses the variable `x` at the same time together with the first assignment. URBI has several *blending modes* to handle these conflicts, that you can specify with the `blend` property of the variable. For example:

```
x->blend = add;
```

This will tell URBI to *add* the numerical values of any conflicting assignments. So, the result of the above command will be 6. There is also a `mix` mode available, which does an average of conflicting assignments (the result would be 3) and a `queue` mode which will queue conflicting assignments (the result will be 5). Other blending modes are available and described in the URBI Language Specification.

Note that blending modes also apply for sound devices, like `speaker` on the Aibo, and changing its blending mode from `mix` to `queue` will either superimpose sounds or queue them when they are played together.

The `add` and `mix` modes are very useful to superimpose sinusoidal assignments to design complex periodical movements, using a Fourier transform of the signal and keeping only the most significant coefficients.

2.7 Useful device variables and properties

For the Aibo robot, and for most standard robots, you will find the following motor device variables useful (replace `device` by the actual device name):

- `device.load` : sets the torque power in a joint, between 0 (totaly loose) and 1 (rigid).
- `device.PGain` : set the P gain of a joint in the associated PID.
- `device.IGain` : set the I gain of a joint in the associated PID.
- `device.DGain` : set the D gain of a joint in the associated PID.

You also have useful *properties*, which are not variables in a strict sense, but you can read and set them:

- `device->rangemin` : minimal value of the device
- `device->rangemax` : maximal value of the device
- `device->delta` : precision of the device
- `device->unit` : unit of the device (for information only)
- `device->blend` : the device blend mode (`normal, mix, add, queue, discard, cancel`)

2.8 Useful commands

Here is a short list of useful commands that you might need in your URBI programs:

- `reset` : does a virtual software reboot of the robot. Useful to erase a set of scripts and send a new version in the development stage
- `stopall` : stop all commands in every connections. A bit radical, but useful sometimes

- `undefall` : erase all variables and functions definition
- `reboot` : reboot the robot
- `shutdown` : stops the robot
- `ping` : echo a pong to make sure the robot is awake
- `uservars` : display a list of the user variables
- `devices` : display the list of known devices
- `defcheckon` : start the strict variable definition control policy (see the URBI Language Specif)

Chapter 3

More advanced language features

At this point, you are already capable of reading and setting sensors and motors in your robot, execute complex scripts or actions and superimpose motion patterns. This could be already enough for most users, but there is more in URBI and the URBI language gives you access to all the programming constructs found in modern languages plus other new constructs useful for robotics.

3.1 Branching and looping

In general, the branching and looping constructs of C/C++ are also available in URBI: `if else`, `for`, `while`. The following examples illustrates these constructs (the `echo` command that you will see simply displays the expression as a system message).

3.1.1 if

`if` performs a single test and execute the associated command if the test is true:

```
if (backSensorM > 0) {
  pressed = 1;
  echo "Back sensor pressed";
}
```

Note that the last command between brackets doesn't need to be ended by a semicolon like in the above example. This is because semicolons are command separators and not terminators. You can put a semicolon at the end like in C, but it is not required and it has not effect (it adds an empty command).

```
if (distance < 10)
  echo "Obstacle detected"
else
  echo "No obstacle";
[167322:notag] *** No obstacle
```

Note that, unlike in C, there is no semicolon before `else`.

`distance` and `backSensorM` are two Aibo devices related to the head infrared distance sensor and to the middle (M) back sensor.

3.1.2 while

The `while` construct is similar to what is available in C:

```
i=0;
while (i<=2) {
  i:echo i;
  i++;
}
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

3.1.3 for, foreach

The `for` construct is similar to what is available in C:

```
for (i=0;i<=2;i++)
  i:echo i;
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

Unlike in C, URBI has specific constructs to handle parallel and serial loops: `for&`, `for|` and `while|`. These constructs will start every iteration in parallel (with `&`) or in series (with `|`) with a guaranteed time constraint. More details are available in the URBI Language Specification.

As we already mentioned before, there is also a `foreach` construct to iterate lists:

```
foreach i in [0,1,2] {
  i:echo i;
}
[151228:i] *** 0
[151228:i] *** 1
[151228:i] *** 2
```

Even when the iterated command is a single command, like in the above example, you must enclose it between brackets.

3.1.4 loop, loopn

For practical reasons, URBI has added two more constructs, `loop` and `loopn` to create infinite loops in the first case and loops iterating n times in the second case. The syntax is:

```
loop { ... }
```

and

```
loopn (n) { ... }
```

3.2 Event catching mechanisms

3.2.1 at

`at` works a bit like `if`, expect that is it always running in the background:

```
at (distance < 50)
  echo "Obstacle appears";
```

The `echo` command in the above example will start at the time when the test becomes true, only once. To be more precise, `at` triggers the command when the test switches from false to true. It is very useful to start an action when a condition is met to react to this condition. If you run the above code, the message "Obstacle appear" will be displayed once when you move your hand in front of the Aibo.

`onleave` is a bit like `else` and is followed by an action that will be executed when the test switches from true to false:

```
at (distance < 50)
  echo "Obstacle appears"
onleave
  echo "The obstacle is gone";
```

3.2.2 whenever

`whenever` works a bit like `while`, except that it never terminates and run in the background:

```
whenever (distance < 50)
  echo "There is an obstacle";
```

The `echo` command will be executed whenever the test is true. Then, it will be executed again if the test is still true, and so on, until the test becomes false. Whenever the test switches to true again, the loop restarts and the command is executed. Compared to the `at` example above, the difference is that the message "There is an obstacle" will be displayed several times, as long as you leave your hand near the head of the robot.

Alternatively, you also have an `else` construct available to specify something to do when the test is false:

```
whenever (distance < 50)
  echo "There is an obstacle"
else
  echo "There is no obstacle";
```

`whenever` and `at` are the two fundamental constructs that you will use when doing reactive programming and event catching mechanisms on your robot.

3.2.3 wait, waituntil

The command `wait` (n) will wait for n milliseconds before ending. It is useful to have a pose in a series of commands, typically motor commands:

```
headPan = 0 |
wait(1000) |
headPan = 90;
```

The command `waituntil`$(test)$ waits until the test becomes true.

3.2.4 timeout, stopif, freezeif

The command `timeout` (n) `cmd` will execute the command `cmd` and stops it after n milliseconds if it is not already finished.

```
timeout(10s) loop legRF2 = legLF2;
```

The command `stopif` $(test)$ `cmd` will execute the command `cmd` and stops it when the $test$ becomes true. Of course, if the command is already finished, nothing special happens.

```
stopif(distance<50) robot.walk();
```

The command `freezeif` (*test*) `cmd` will execute the command `cmd` and freezes it when the *test* becomes true. When the test is false again, `cmd` is unfreezed.

```
freezeif(!ball.visible) trackball();
```

3.2.5 Soft tests

The tests used in event catching commands like `at`, `whenever`, `waituntil`, `stopif` or `freezeif` can be associated to time constraints, or "soft tests":

```
at (headSensor >0 ~ 2s)
  echo "Head touched...";
```

This means that the test has to be true for 2 seconds before it becomes actually true for the `at` command. You can specify the time in *s* or *ms* by using the appropriate suffix and it is separated from the test by a tilde ~

Soft tests are usable with any event catching command and they are very useful in robotics as a simple noise filter for sensor inputs.

3.2.6 Emit events

Event programming is a very cool feature and a good way of doing robot programming. The basic idea of event programming is that some command emit an event and some other catches this event and do something.

Simple events

To emit an event, the `emit` command is available in URBI, and you can use `at` or `whenever` to catch it:

```
at (boom) echo "boom!";
emit boom;
[139464:notag] *** boom!
```

Note that the `boom` event here is local to the connection. If you want to make the event visible from other connection, you should use a prefix, like `myprefix.boom`.

Events with parameters

You can add parameters to events like this:

```
emit myevent(1,"hello");
```

The parameters can be retrieved when the event is catched:

```
at (myevent(x,y))
  echo "catch two:  " + x + " " + y;

at (myevent(1,x))
  echo "catch one:  " + x;
```

The second `at` here is doing a filtering on the event parameters, accepting only events whose first parameter equals 1:

```
emit myevent(1,"hello");
[146711:notag] *** catch two:  1.000000 hello
[146711:notag] *** catch one:  hello
emit myevent(2,15);
[148991:notag] *** catch two:  2.000000 15.000000
```

Event duration

Events usually have a virtually null duration, they are just spikes (Dirac functions). You can explicitly requests that an event lasts for a certain duration by specifying this duration between parenthesis like this:

```
emit(10s) boom;
emit(15h12m) myevent(1,"hello");
```

This will make a difference between `at` and `whenever` event catcher for example.

Pulsing events: the `every` command

You can have any command repeated at specific time intervals in URBI, using the `every` command. The following example will say "hello" every 10 minutes:

```
every (10m) echo "hello";
```

You can use the `every` command to pulse events at regular intervals:

```
every (100ms) emit pulse;
```

To stop the emission, just use `stop` with the appropriate tag (see 3.3):

```
mypulse:every (100ms) emit pulse;
stop mypulse;
```

3.3 Command flags and command control

The tagging mechanism described in the beginning of this tutorial is actually more than just a message tagging facility. For example, you can stop any running command with the `stop` command, from any connection:

```
myloop:loop legRF2 = legLF2,
...
stop myloop;
```

You can also freeze a command with the `freeze` command and unfreeze it (it will restart where it was before freezing) with the `unfreeze` command. There is also a `block/unblock` pair of commands to block new commands with a given tag and prevent

them to be executed. Note that tags can prefix a set of commands between brackets, like { ... }, and it can be associated to large portions of code, not only single commands.

Next to the tag, it is possible to use one or more *flags*. Flags are keywords prefixed by a + sign. The most useful tags are `+begin` and `+end` which send a system message when the command starts or stops or `+bg` which puts the command in background. Here are a set of illustrating examples:

```
mytag+begin:
  loop legRF2 = legLF2,
[139464:mytag] *** begin
```

```
+begin+end:
  wait(1s);
[521200:mytag] *** begin
[522200:mytag] *** end
```

More flags are described in the URBI Language Specification.

3.4 Device grouping

An important feature of URBI is the capacity to group devices into hierarchies. This is done with the `group` command: `group virtualdevice { device1, device2, ...}`, for example:

```
group legLF {legLF1, legLF2, legLF3};
group legs {legLF, legLH, legRF, legRH};
```

This grouping feature is used to make *multi-device assignments*: any assignment is executed for the device and then it is recursively passed to child subdevices. In other words, using the example above, the command `legLF = 0` will set the value of `legLF1`, `legLF2` and `legLF3` to 0.

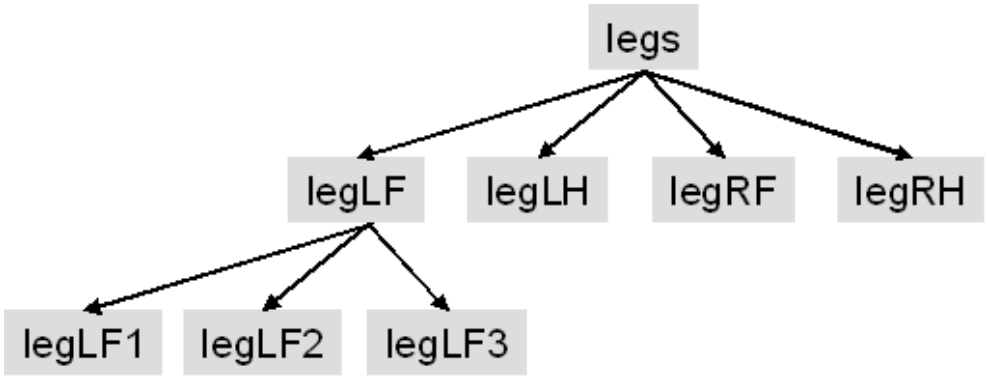

Figure 3.1: A typical motor device hierarchy

```
group ab {a,b};
ab.n = 4;
a:a.n, b:b.n, ab:ab.n;
[167322:a] 4.000000
[167322:b] 4.000000
[167322:ab] 4.000000
```

For any robot, there will usually be a hierarchy of devices available at start. This is usually done in the `URBI.INI` file, or `std.u`.

For example, with the Aibo, there is a `leds` virtual device which contains all led devices and you can easily set every LEDs to a random value with a command like this:

```
leds = random(2);
```

For fun, you can run something like: `loop leds = random(2),` and see the result.

3.5 Function definition

Like in any advanced language, functions can be defined in URBI. To do so, you will be using the `def` keyword, followed by the function name in `prefix.suffix` notation (or simply `suffix` for a function local the connection), and the parameters between brackets (or an open/close bracket () if there is no parameters). You can use `return` to return a value or to exit from the function, like in C:

```
def adding(x,y) {
  z = x+y;
  return z;
};
```

```
def print(x) {
  echo x;
  if (x<0) return
  else
    echo sqrt(x);
};
```

Note that there must be a semicolon or another command separator after the function definition, since defining a function is a command like any other command in URBI.

The parameters are always local to the function call. Non-global (i.e. without prefix) variable in the function body are also local to the function call. Consider the following example:

```
a=4;
b=5;
def display(b) {
  display_b:b; // b is local
  a=b; // creates a local variable a
  display_a:a;
};
display(10);a:a;b:b;
[139464:display_b] 10.000000
[139464:display_a] 10.000000
[139464:a] 4.000000
[139464:b] 5.000000
```

A good idea is to put all your functions in a separate file like `"myfunc.u"`, and load them with the `load` command: `load("myfunc.u");` This can be done from the `URBI.INI` file for example, or when you actually need them.

3.6 Error messages and system messages

When a command fails in URBI, it will send an error message, prefixed by three exclamation marks:

```
impossible:1/0;
[167322:impossible] !!!  Division by zero
[167322:impossible] !!!  EXPR evaluation failed
```

Note that the tag of the command is used in the error message, which is extremely convenient to know what has failed in a complex program.

Error messages are different from system messages prefixed by three stars, and which usually display information normally outputted or requested by the command. A typical example is `echo` with a `+begin` and `+end` flag:

```
mytag+begin+end:echo "hello there!";
[146711:mytag] *** begin
[146711:mytag] *** hello there!
[146711:mytag] *** end
```

Chapter 4

The ball tracking example

The best way to learn a new language is to study simple examples to see what can be done in practice. In this tutorial, we will concentrate on the red ball tracking application of Aibo which is interesting because it is a simple behavior with two states and it involves a perception/action loop which is very typical of robotic applications. We will see how URBI can help to control the execution of the behavior in a simple way with command tags.

4.1 Ball detection

Detecting a ball involves image processing and cannot be written directly in URBI for obvious efficiency reasons. The best way to provide such components (like visual processing or sound processing) is to write a *soft device* in C++, Java or Matlab, and to interface it in URBI. We will not describe at this stage how to write such a "soft" device, but instead we will already use one: the `ball` soft device.

The `ball` soft device is directly integrated in the Aibo URBI server and you can use it directly, just like any physical device. It has no `ball.val` variable but it has a `ball.x` and `ball.y` variable which are equal to the coordinate of the ball in the image between $-1/2$ and $1/2$. When there is a ball visible, `ball.visible` is equal to 1, zero otherwise. It also have a `ball.size` variable which give the size of the ball in the image, expressed as a number of pixels. These simple soft device variables are already enough to do many interesting applications, as we will see below.

4.2 The main program

The ball tracking program is given as an example in the Sony SDK (OPEN-R) and does the following: when there is a ball in front of the robot, it will track it by moving the head in the ball direction, otherwise it will scan the surrounding environment by moving the head in circles.

Moving the head in the direction of the ball can be written very simply in URBI with these two lines of code:

```
headPan  = headPan  + camera.xfov * ball.x &
headTilt = headTilt + camera.yfov * ball.y;
```

The effect is to move at the same time the head motors in pan and tilt directions, by an amount proportional to the `x` and `y` position of the ball in the image. The `camera.xfov` and `camera.yfov` coefficients are coming from the `camera` device that we will discover in the next chapter. They represent the x-angular and y-angular field of view of the Aibo camera, which are used here to convert the $[-1/2; 1/2]$ segment of `ball.x` and `ball.y` into actual angles in degrees.

To actually track the ball and not simply move once in its direction we will use a `whenever` command:

```
whenever (ball.visible) {
  headPan  = headPan  + camera.xfov * ball.x &
  headTilt = headTilt + camera.yfov * ball.y;
};
```

This program is only three lines long and does the ball tracking behavior expected. However, on the Aibo, it might be to reactive and lead to small oscillations of the head around the ball position. To avoid this, a simple technics from robotic control is to use an attenuation coefficient, `ball.a`, to limit the reactivity of the system. For example:

```
ball.a = 0.8;
whenever (ball.visible) {
  headPan  = headPan + ball.a * camera.xfov * ball.x &
  headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};
```

The next step is to switch from this behavior to the scanning behavior when the ball is not visible. The scanning behavior can be expressed with a simple sinusoidal movement on both `headPan` and `headTilt`. We use in the following example the `'n` variable extension[1] which indicates that we are working with the normalized value of the variable, between 0 and 1, calculated from the known `rangemin` and `rangemax`. It is very convenient to avoid checking the actual range of a device and use it in a more general way:

```
period = 10s;
headPan'n  = 0.5 sin:period ampli:0.5 &
headTilt'n = 0.5 cos:period ampli:0.5,
```

The `cos` modifier is identical to `sin` with a phase shift of $pi/2$. Note how the central value of 0.5 with the amplitude of 0.5 allows to cover the full range of the device: $[0..1]$

The above command does the circular movement required but the first position in the circle will be reached abruptly from wherever the head was before the command starts. To avoid this, we can precede the command with a smooth transition in one second towards the initial position in the circle which is `headPan'n = 0.5` and `headTilt'n = 1`:

```
headPan'n  = 0.5 smooth:1s &
headTilt'n = 1 smooth:1s;
```

The `smooth` modifier is similar to `time` but with a smooth movement, instead of a linear movement.

Now, we can connect everything into the whole behavior. To avoid switching from the circular sweeping to the ball tracking to often, we also add a soft test, and we use the `loadwav` function to preload two wav files that we assign to the `speaker` device (described later) to play a sound when the ball is found or lost:

```
// Parameters initialization
ball.a = 0.9;
period = 10s;
found = loadwav("found.wav");
lost = loadwav("lost.wav");
```

[1]Other extensions are available in URBI. Extensions are a powerful way to modulate the evaluation of a variable. Check the URBI Language Specification for more details

```
// Main behavior
whenever (ball.visible ~ 100ms) {
  headPan  = headPan + ball.a * camera.xfov * ball.x &
  headTilt = headTilt+ ball.a * camera.yfov * ball.y;
};

at (!ball.visible ~ 100ms)
  search:  {
    { headPan'n  = 0.5 smooth:1s &
      headTilt'n = 1 smooth:1s } |
    { headPan'n  = 0.5 sin:period ampli:0.5 &
      headTilt'n = 0.5 cos:period ampli:0.5 }
  };
at (ball.visible) stop search;

// Sound behavior
at (ball.visible ~ 100ms) speaker = found
onleave speaker = lost;
```

You can also use the `onleave` construct to group the two `at (ball.visible)` commands, but you must use the `at&` command in that case, to put the `search` command in background (because it is a never ending command and `at` would never get the hand again otherwise).

4.3 Programming as a behavior graph

The above program works fine and is easy to understand and maintain. However, it is common in robotics to design programs in terms of behaviors, which are graphs of *states* connected together with *transitions*. Fig. 4.1 illustrates the behavior graph of the ball tracking program, which is an very simple example of a two states behavior.

The ellipses represent states (in which the robot is doing some basic action/perception loop) and the arrows are the transitions, expressed as conditions. The squares attached to the transition specify some action to trigger when the transition occurs.

The best way to program this kind of behavior graph in URBI is to use a conjunction of functions, `at` and `stop` commands. First, let's define the two functions related to the two states of the ball tracking program:

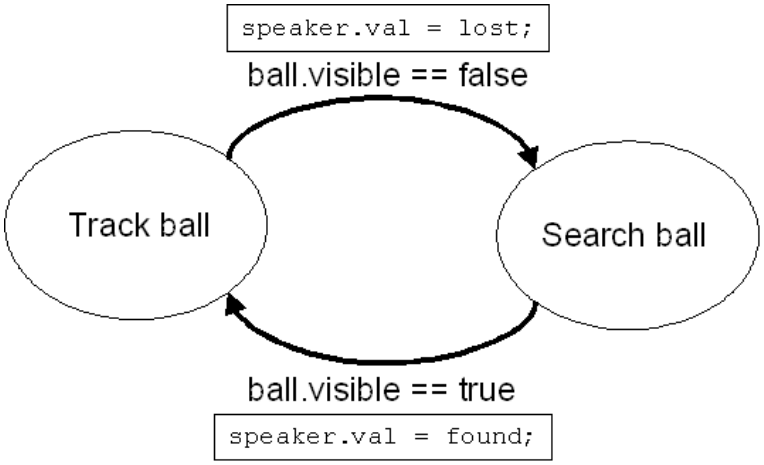

Figure 4.1: The ball tracking behavior graph

```
// Tracking state
def tracking() {
  whenever (ball.visible) {
    headPan  = headPan + ball.a * camera.xfov * ball.x &
    headTilt = headTilt+ ball.a * camera.yfov * ball.y;
  }
};

// Searching state
def searching() {
  period = 10s;
  { headPan'n  = 0.5 smooth:1s &
    headTilt'n = 1 smooth:1s } |
  { headPan'n  = 0.5 sin:period ampli:0.5 &
    headTilt'n = 0.5 cos:period ampli:0.5 }
};
```

Now, we can simply "glue" the states together by stating the transitions as two `at` commands with `stop` commands to terminate the previous state:

```
// Transitions
at (ball.visible ~ 100ms) {
  stop search;
  speaker = found;
  track:  tracking();
};

at (!ball.visible ~ 100ms) {
  stop track;
  speaker = lost;
  search:  searching();
};
```

The advantage of rewriting the ball tracking program in terms of behavior may not appear very clear at this stage, because the program is very simple. However, with more complex behaviors including tens of different states, each with several transitions, this is the best and safest way to program. It makes the code modular, clear and easy to maintain.

4.4 Controlling the execution of the behavior

The possibility to freeze, stop or block commands in URBI is a very powerful tool to control the execution of a behavior. For example, if the transitions expressed with a `at` command are prefixed by a tag, like this:

```
track_transition:  at (ball.visible ~ 100ms) {
  stop search;
  speaker = found;
  track:  tracking();
};

search_transiton:  at (!ball.visible ~ 100ms) {
  stop track;
  speaker = lost;
  search:  searching();
};
```

It becomes very easy to activate or temporarily suspend a transition by commands like:

```
freeze track_transition;
...
unfreeze track_transition;
```

Also, it is possible to block the execution of a state, but still accept transitions to this state (waiting silently for another transition to make the robot move to another state):

```
block search;
...
unblock search;
```

Using `freeze`, `block` and `stop`, it is simple to modify behaviors or reassign priorities online during the execution of a program, which is a very useful feature for robotics.

Chapter 5

Images and sounds

Until now, we have only used numerical devices, like `headPan`. This, of course, is not sufficient to transmit images or sounds. Some devices, like for example `camera`, `micro` or `speaker` in Aibo, are *binary devices*. In that case, the `device.val` variable is not a numerical value but a binary value.

5.1 Reading binary values

You might have already tried to evaluate one of those binary variables:

```
camera;
[139464:notag] BIN 5347 jpeg 208 160
.................5347 bytes.................

micro;
[139464:notag] BIN 2048 wav 2 16000 16 1
.................2048 bytes.................
```

URBI simply prefixes the binary data with a header starting with the keyword `BIN`, followed by the size (in octets) and a keyword indicating the type of the data. Optional parameters, like the size of the image or the sampling rate and stero/mono status of a sound might follow. Then, after a carriage return, the actual binary data is returned (displayed above as ....), which might confuse a telnet client but not a software client or URBILab[1].

[1] URBILab understands URBI headers and display images or play sounds according to the type

What we call a "software client" is a client written in a language like C++ or Java, as described in detail in chapter 6. This is the normal way of handling binary data when you want to do complex signal processing with URBI.

5.2 Setting binary values

As you might expect, setting a binary value into a speaker device for example is not more complex than reading it. To play a sound on Aibo, you could send to the server a command like this:

```
speaker = bin 54112 wav 2 16000 16;
..............54112 bytes.............
```

It is important that the header ends with a semicolon. The binary content starts immediately after the semicolon, so you don't have to add an extra carriage return.

Of course, as we already said it, this kind of binary assignment will obviously not be done from a telnet or URBILab client, since you probably want that a program sends the binary content, and you cannot type it yourself. However, we will see in the next section how you could simply play a sound from a telnet client if you need to.

This simple example illustrates a binary assignment and a binary reading in URBI from a telnet client, however it is a "toy" example:

```
mybin = bin 3;ABC
mybin;
[146711:notag] BIN 3
ABC
```

Note that you can pass any parameters after the size of the binary data and they will be stored together with the binary content, inside the header:

```
mybin = bin 3 hello world 33;ABC
mybin;
[146711:notag] BIN 3 hello world 33
ABC
```

Do not confuse binary data and string data. The above example is different from:

```
mystring = "ABC";
mystring;
[148991:notag] "ABC"
```

5.3 Associated fields

Usually, with a binary device you have a set of associated fields available. A typical example is the `camera` device which provides the following fields on Aibo:

- `camera.shutter` : the camera shutter speed: 1=SLOW (default), 2=MID, 3=FAST
- `camera.gain` : the camera gain: 1=LOW, 2=MID, 3=HIGH (default)
- `camera.wb` : the camera white balance: 1=INDOOR (default), 2=OUTDOOR, 3=FLUO
- `camera.format` : the camera image format: 0=YCbCr 1=jpeg (default)
- `camera.jpegfactor` : the jpeg compression factor (0 to 100). Default=80
- `camera.resolution` : the image resolution: 0:208x160 (default) 1:104x80 2:52x40
- `camera.reconstruct` : reconstruction of the high resolution image(slow): 0:no (default) 1:yes
- `camera.width` : image width
- `camera.height` : image height

- `camera.xfov` : camera x Field Of View (degrees)
- `camera.yfov` : camera y Field Of View (degrees)

In the case of the `speaker` device, in charge of the speaker producing sound in the Aibo, you have:

- `speaker.playing` : equal 1 when there is a sound playing, 0 otherwise
- `speaker.remain` : number of milliseconds of sound to play, 0 when the buffer is empty.

With the `speaker` device, there is also a method that can be used to play a sound directly from a file stored on the memorystick:

```
speaker.play("mysound.wav");
```

Alternatively, to avoid having a disk access which might be slow, you can decide to store the content of the "mysound.wav" file in a binary variable kept in memory for frequent use, and then do a simple assignment. For this, use the `loadbin` function:

```
mybin = loadwav("mysound.wav");
speaker = mybin;
```

5.4 Binary operation examples

There is a possibility in URBI to add binaries, which is mainly used for sound concatenation. For example, consider the following program:

```
sound = bin 0;
timeout(10s) loop sound = sound + micro;
speaker = sound;
```

This code will record 10 seconds of sound from the `micro` device and store it in the `sound` variable, and then will play it back be assigning `sound` to the speaker device. It shows how simple it can be to manipulate binary buffers with URBI for simple tasks like concatenation.

Chapter 6

The liburbi in C++

6.1 What is liburbi?

Using URBI with a telnet client is too limited. You need to be able to send commands and receive messages using a programming language of your choice.

That's why we call URBI an "Interface Language": it's more than a simple protocol because it's a full featured script language but it is less than a script language that stands on its own, because in most applications where you have computer vision of sound processing, you use URBI together with C++ or another fast language to do the algorithmic part. URBI is here to run the architecture of your behaviors, your action/perception loops and other high level elements, using the output of the fast C++/Java/Matlab client as inputs for its decisions.

What is liburbi? You could program a TCP/IP layer for C++ or for your favorite language but this is trivial and should be done once only. This is what liburbi is made for. What you want to be able to do are things like:

- Open a connection to your robot from within your favorite language (like C++)
- Send a command to your robot from within that language
- Ask for a device value and receive it
- Listen to incoming messages from your robot and react to them appropriately

Actually, the last point is the most important and, however it might differ from the way you may be used to write programs, it is essential to adapt to this way of thinking (called "asynchronous programming") because it is best suited for robotics. Robots are fundamentally asynchronous

devices. You want to wait for messages from your robot and react to them (it's also called "event-driven programming"). A that's what a robot does most of the time: react to events[1].

This chapter is a brief introduction to liburbi. You should read the official liburbi documentation on www.urbiforge.com if you want a comprehensive description.

6.2 Software modules and liburbi

Making software components written in C++, Java or Matlab available for your URBI scripts can be done in two different ways. The first way it to use one of the liburbi flavor for your preferred language (C++/Java/Matlab). That is what we are going to describe in this chapter.

The second option, described in chapter 7 is to create a *software device*, accessible like any other device from your URBI server, with methods and device variables. It is the most portable and flexible way of adding functionalities to URBI, but let's start with the basic liburbi. In any case, knowing the liburbi approach is a good idea, since it might be more suited to your need in certain cases and it gives a good introduction to asynchronous programming.

There is currently a C++, Matlab and Java version of liburbi if you want to control your robot using C++, Matlab or Java and a liburbi-OPENR version if you want to recompile a liburbi-C++ based program to let it run on the Aibo, as an OPENR object (in that case, your robot will remain completely autonomous). We will not describe all those possibilities but only the C++ version, which gives the general ideas. Other versions are similar and have a specific documentation. We assume in the following that you have a basic understanding of C++. If not, please refer to a simple C++ tutorial, since the notion developed here will remain basic.

6.3 Getting started

To start with, you need to be able to compile a liburbi-based program. There are several ways to do so, depending on the fact that you are on Linux or Windows, with Borland or Microsoft compiler, etc. In general, you are simply supposed to include `liburbi.h` and link your code with the `"-lurbi"` parameters (gcc) or similar syntax. See the appropriate documentation for more details.

Then, the first thing you need to do is create a client connected to your robot "myro-

[1]Traditionally in AI, the way the robot reacts might by modified by higher level cognitive activities (hierarchical architecture) or by priorities (subsumption architecture) or by a complex combination of deliberative and reactive processes (hybrid architecture)

bot.mydomain.com". For this purpose, you have a `UClient` class in liburbi-C++:

```
UClient* client = new UClient("myrobot.mydomain.com");
```

If you have only an IP address, you can use it instead of the server name.

Alternatively, you might want to use the `urbi` namespace instead:

```
UClient* client = urbi::connect("myrobot.mydomain.com");
```

Of course, you can create as many clients as you like with this method.

6.4 Sending commands

The `UClient` object has a `send` method which works like `printf`:

```
client->send("motor on;");
for (float val=0; val<=1; val+=0.05)
  client->send("neck'n = %f;wait (%d);", val, 50);
```

You can also use your client object as a stream if you prefer a more C++ like approach:

```
client << "headPan = " << 12 << ";" ;
```

There is also a very cool way of sending blocks of URBI code from a C++ program, using the `URBI((...))` macro:

```
URBI((

  headPan = 12,
  echo "hello" | speaker.play("test.wav") & leds = 1

));
```

The text between the double parenthesis will be sent to the first client created by your program, by default. This can be set with a call to `urbi::connect(...)`. The first method, using the `send` method is more appropriate in general and the `URBI` macro should only be used to send initialization scripts in a convenient way at the beginning of your program.

Remember that you can always give your robot a fresh start (a virtual reboot) by sending the `reset` command. This will avoid to multi definition of functions or restarting several occurrences of an `at` command each time you rerun your liburbi-based client. So most liburbi main programs will start with `client->send("reset;");`

6.5 Sending binary data and sounds

To send binary data, you will use the `sendBin` method, instead of `send`:

```
client->sendBin(soundData, soundDataSize,
     "speaker = BIN %d raw 2 16000 16 1;",
     soundDataSize);
```

The first two parameters are the binary data itself and the size. Then, the header, with optional parameters with a `printf` like syntax.

To send a sound, there is specialized method called `sendSound`, which is more convenient and also more efficient:

```
client->sendSound(sound, "endsound");
```

The first parameter is a `USound` structure (see 6.7.2), describing the sound to send. The second is an optional tag that will be used by the server to issue a "stop" system message when the sound has finished playing.

The function `convert` described in the documentation can be used to convert between various sound formats.

With `sendSound`, there is no limit to the size of the sound buffer, since it will be automatically cut into small chunks by the library. Since the data is copied by liburbi, the `USound` parameter and its associated data can be safely freed as soon as the function returns.

6.6 Receiving messages

URBI tags are going to prove very useful for receiving incoming messages from the server: each command has an associated a tag (`notag` by default), and this tag is repeated in any message originating from this command. The `UClient` class handles the reception of those messages in an independent thread created by the constructor, parses them and fills a `UMessage` structure (see 6.7.1). Then, callback functions with the associated tag can be registered with the method `setCallback`: each time a message with this tag is sent by the server, the callback function will be called with the `UMessage` structure as a parameter.

```
typedef UCallbackAction (*UCallback) (const UMessage &msg);

UCallbackID setCallback (UCallback cb, const char *tag)
```

The first parameter `cb` is a pointer to the function to call. The callback function must return URBI_CONTINUE, or URBI_REMOVE, in which case the function will be unregistered.

The best way to learn about how callbacks can be used with the liburbi is to look at some example, like the one described in the liburbi documentation page at:

`http://www.urbiforge.com/eng/liburbi.html`

6.7 Data types

The data type used by the liburbi are described below:

6.7.1 UMessage

```
class UMessage {
 public:

  UAbstractClient    &client;   // connection from which originated the message
  int                timestamp;     // server-side timestamp
  char               *tag;          // associated tag

  UMessageType       type;          // type of the message
  UBinaryMessageType binaryType;    // type of binary message

  union {
    double        doubleValue;
```

```
    char          *stringValue;
    char          *systemValue;
    char          *message;        // filled if type is unknown (MESSAGE_UNKNOWN)
    USound        sound;           // filled if binary data is of the sound type
    UImage        image;           // filled if binary data is of the image type
    UBinary       binary;          // filled if binary data is of an unrecogn. type
  };
}
```

The type field can be MESSAGE_DOUBLE, MESSAGE_STRING, MESSAGE_SYSTEM, MESSAGE_BINARY or MESSAGE_UNKNOWN. Depending of this field, the corresponding value in the union will be set. If the message is of the binary type, binaryType will give additional informations on the type of data (BINARYMESSAGE_SOUND, BINARYMESSAGE_IMAGE or BINARYMESSAGE_UNKNOWN), and the appropriate sound or image structure will be filled.

6.7.2 USound

```
class USound {
 public:
 char                  *data;            // pointer to sound data
 int                   size;             // total size in byte
 int                   channels;         // number of audio channels
 int                   rate;             // rate in Hertz
 int                   sampleSize;       // sample size in bit
 USoundFormat          soundFormat;      // format of the sound data
                                            //(SOUND_RAW, SOUND_WAV, SOUDN_MP3...)
 USoundSampleFormat    sampleFormat;     // sample format
};
```

6.7.3 UImage

```
class UImage {
 public:
  char                  *data;            // pointer to image data
  int                   size;             // image size in byte
  int                   width, height;    // size of the image
  UImageFormat          imageFormat;      // IMAGE_RGB, IMAGE_YCbCr, IMAGE_JPEG...
};
```

6.8 Synchronous operations

The derived class `USyncClient` implements methods to synchronously get the result of URBI commands. You must be aware that these functions are less efficient, and that they will not work in the OPEN-R version of the liburbi, for instance.

6.8.1 Synchronous read of a device value

To get the value of a device, you can use the method `syncGetDevice`. The first parameter is the name of the device (for instance, `"neck"`), the second is a double that is filled with the received value:

```
double neckVal;
syncClient->syncGetDevice("neck",neckVal);
```

6.8.2 Getting an image synchronously

You can use the method `syncGetImage` to synchronously get an image. The method will send the appropriate command, and wait for the result, thus blocking your thread until the image is received.

```
client->send("camera.resolution = 0;camera.gain = 2;");
int width, height;
client->syncGetImage("camera", myBuffer, myBufferSize,
  URBI_RGB, URBI_TRANSMIT_JPEG, width, height);
```

The first parameter is the name of the camera device. The second is the buffer which will be filled with the image data. The third must be an integer variable equal to the size of the buffer. The function will set this variable to the size of the data. If the buffer is too small, data will be truncated .

The fourth parameter is the format in which you want to receive the image data. Possible values are `URBI_RGB` for a raw `RGB` 24 bit per pixel image, `URBI_PPM` for a `PPM` file, `URBI_YCbCr` for raw `YCbCr` data, and `URBI_JPEG` for a jpeg-compressed file.

The fifth parameter can be either `URBI_TRANSMIT_JPEG` or `URBI_TRANSMIT_YCbCr` and specifies how the image will be transmitted between the robot and the client. Transmitting `JPEG` images increases the frame rate and should be used for better performances.

Finally the width and height parameters are filled with the with and height of the image on return.

6.8.3 Getting sound synchronously

The method `syncGetSound` can be used to get a sound sample of any length from the server.

```
client->syncGetSound("micro", duration, sound);
```

The first parameter is the name of the device from which to request sound, the second is the duration requested, in milliseconds. Sound is a `USound` structure (see 6.7.2) that will be filled with the recorded sound on output.

6.9 Conversion functions

We also have included a few functions to convert between different image and sound formats. The usage of the image conversion functions is pretty straightforward:

```
int convertRGBtoYCrCb(const byte* source, int sourcelen, byte* dest);
int convertYCrCbtoRGB(const byte* source, int sourcelen, byte* dest);
int convertJPEGtoYCrCb(const byte* source, int sourcelen, byte* dest, int &size);
int convertJPEGtoRGB(const byte* source, int sourcelen, byte* dest, int &size);
```

The size parameter must be set to the size of the destination buffer. On return it will be set to the size of the output data.

To convert between different sound formats, the function convert can be used. It takes two `USound` structures as its parameters. The two audio formats currently supported are `SOUND_RAW` and `SOUND_WAV`, but support for compressed sound formats such as Ogg Vorbis and MP3 is planned. If any field is set to zero in the destination structure, the corresponding value from the source sound will be used.

6.10 The "urbiimage" example

URBIimage is a simple program written in C++ with the liburbi-C++ to get and display images from an URBI server. URBIimage does two things: it sets a callback on a tag named `uimg` and then receives the images in this callback and send them to a display object `Monitor`. Let's have a look at the general code and the `main` function. First, the callback interface:

```
Monitor *mon;

/* Our callback function */
UCallbackAction showImage(const UMessage &msg)
{
...
}
```

Then, the `main` function:

```
int main(int argc, char *argv[])
{
  mon = NULL;

  client = new UClient(argv[2]);
  if (client->error() != 0)
    exit(0);

  client->setCallback(showImage, "uimg");

  // Some image initialization
  client->send("camera.resolution = 0;");
  client->send("camera.jpegfactor = 80;");

  // Start the loop
  client->send("loop uimg:  camera,");
  urbi::execute();
}
```

The code to handle the image is stored in `showImage`:

```
UCallbackAction showImage(const UMessage &msg)
{
  if (msg.binaryType != BINARYMESSAGE_IMAGE)
    return URBI_CONTINUE;

  unsigned char buffer[500000];
  int sz = 500000;
  static int tme = 0;

  if (!mon)
    mon = new Monitor(msg.image.width, msg.image.height);

  convertJPEGtoRGB((const byte *) msg.image.data,
    msg.image.size, (byte *) buffer, sz);

  mon->setImage((bits8 *) buffer, sz);
  return URBI_CONTINUE;
}
```

It first tests for the `msg` type, and returns without doing anything if this is not the `BINARYMESSAGE_IMAGE` type expected (for example, if the callback is waken up by an error message).

Then, the conversion function `convertJPEGtoRGB` is used to transform the image buffer in something readable for the `Monitor` object, which then receives the image.

Finally, `URBI_CONTINUE` is returned to carry on receiving future callbacks.

This little program illustrates very well how a liburbi-based URBI program is built: set callbacks, send URBI scripts, receive callbacks in specified functions. You might have a look at the GPL source code of URBILab which is built with liburbi-C++ and shows a more advanced use of this methodology.

Chapter 7

Create soft devices

We will soon describe here how to write a software component called "soft device". An example would be a voice synthesis/recognition soft device called `voice` which would emit `voice.hear(s)` event when it recognizes a word "s" and which would synthesize and play any given message received via the `voice.say(s)` function. Here is an example:

```
at (voice.hear("hello")) {
  echo "I've heard hello!";
  ...
}

voice.say("I'm a happy robot");
```

For the moment, only the `ball` soft device is available in Aibo. Its job is to keep the `ball.x`, `ball.y` and `ball.size` variables up to date.

Other functions like controlling the soft device activity (on/off) or priority (nice level) will also be available. But the most important feature is that it will be the same C++ soft device code to make the soft device running inside the URBI kernel, outside on a remote computer or onboard next to the server. This will allow a great deal of flexibility.

Chapter 8

The "ball" soft device example

To be coming soon...

Chapter 9

Putting all together

The following diagram (fig. 9.1) shows a typical setting of clients and software architecture for an URBI application. You have clients in C++, Java and Matlab running on different machines (with Linux, Windows, Mac OSX) plus onboard clients and some telnet/URBILab scripting. There is also a bench of controlling scripts running from the URBI.INI file. This example shows how flexible URBI can be, having all those systems running in parallel to control your robot.

Typical usages examples

You have an URBI server running on your robot and...

1. Remote control of a robot

   Your robot is equipped with a wifi connection, like Aibo, and you run a complex AI program on a powerful desktop computer to control it. This program is actually an URBI client written in C++ and uses the liburbi C++ library to send URBI commands to the robot when needed and to receive URBI messages from the server asynchronously and react to them (asynchronism is achieved with a callback mecanism, see the liburbi doc for more details). You can replace C++ by Java, Matlab or any language you like if you don't want C++.

2. Fully autonomous robot with an onboard URBI client

   This time, you run the URBI client on the robot and not remotely. Just like before, it is written in C++ with the liburbi C++ (or the liburbi-OPENR in the case of Aibo). Instead of a TCP/IP wifi based connection between your client and the server, you have a direct interprocess communication on localhost (or OPENR based message passing in the case of Aibo).

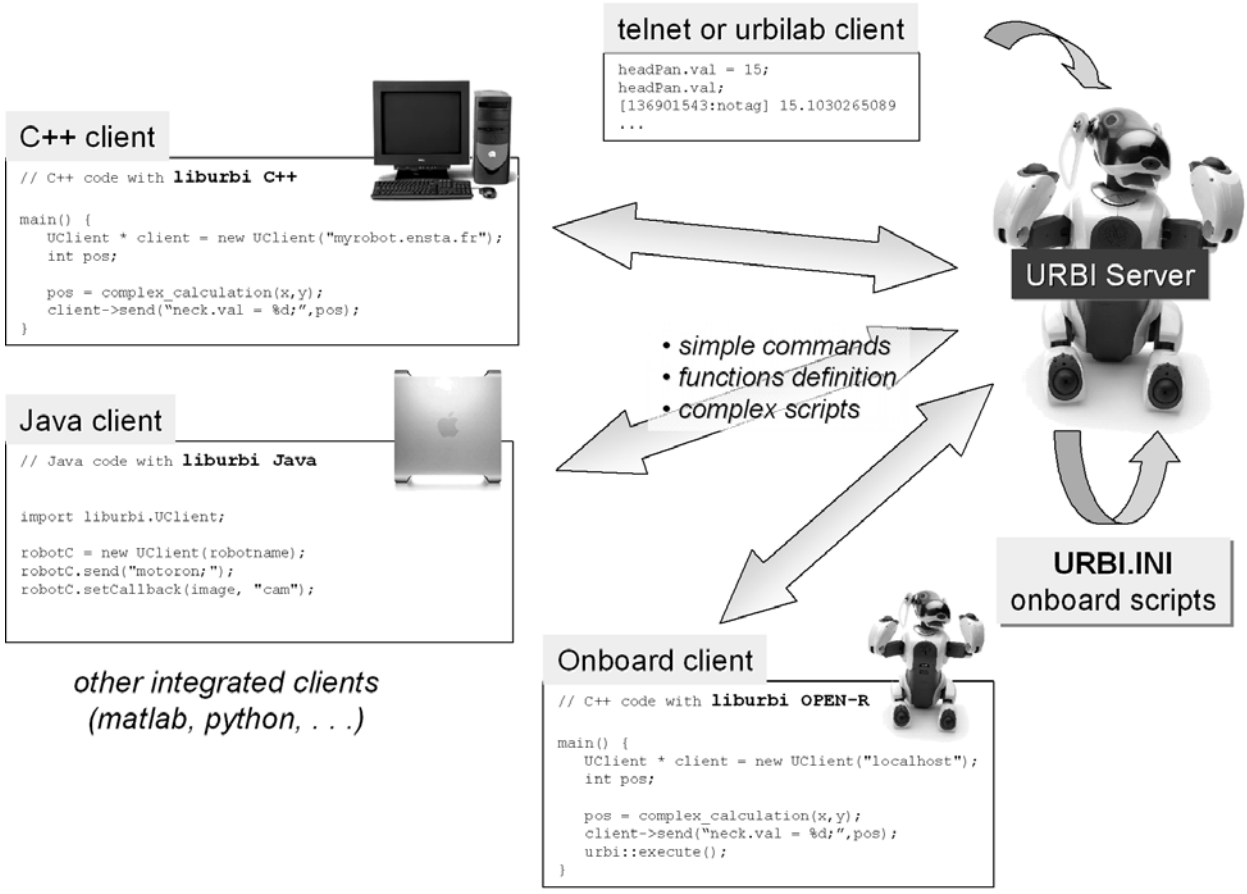

Figure 9.1: The general URBI architecture, putting all together

3. Fully autonomous robot controlled only by URBI scripts

   In that case, it means that you have found all the functionalities you need in URBI (no need for external C++ or Java programming) and you write directly all the action-perception loops with URBI scripts running in the URBI server. You need a simple telnet or URBILab to send those scripts to the server and it is set. You can also store the script directly in the URBI.INI file and your robot will start it at boot up.

4. A mix of 1, 2 and 3

   You have a robot controlled by several URBI clients at the same time, some on the robot, some on a desktop computer, some in C++, some in Java. On top of that, you have several URBI scripts running in the server to perform reactive action-perception loops, started from URBI.INI but also dynamically loaded by some of the clients when needed. This is the most interesting situation, making a full use of the URBI flexibility. See fig 9.1.