

# URBI Language Specification

Official v 1.0

Jean-Christophe Baillie  
©Gostai, 2005-2007

October 2007

# Contents

<b>1</b>	<b>URBI Language Specification</b>	<b>5</b>
1.1	URBI Port	5
1.2	Comments	5
1.3	Commands, Time Operators and Messages	5
1.3.1	Time Operators	6
1.3.2	Tagged commands	7
1.3.3	Flags	7
1.3.4	Grouped commands	7
1.3.5	Messages	8
1.4	Objects	8
1.4.1	Definition	8
1.4.2	Complementary fields and methods	9
1.5	Variables	9
1.5.1	Variables types	9
1.5.2	Variables syntax	10
1.5.3	Variables declaration	10
1.5.4	Variable properties	11
1.5.5	Variables by name	12
1.5.6	Static variable use	12
1.5.7	Variable aliases	13
1.5.8	Variables inside functions	13
1.5.9	Variable extensions	14
1.6	Assignments, Modifiers and blending modes	16
1.6.1	Float assignment	16
1.6.2	String assignment	19
1.6.3	List assignment	19
1.6.4	Binary assignment	20
1.6.5	Blending modes	20
1.7	Expressions	21
1.7.1	Float	21
1.7.2	String	21
1.7.3	List	22
1.7.4	Binary	22
1.7.5	Boolean	22
1.7.6	Standard functions	22
1.8	Control and event catching structures	23
1.8.1	Control structures	23
1.8.2	Soft tests	26
1.8.3	Event catching structures	26
1.9	Emitting events	27

1.9.1	Simple events . . . . .	28
1.9.2	Events with parameters . . . . .	28
1.9.3	Event duration . . . . .	28
1.9.4	Pulsing events: the <i>every</i> command . . . . .	28
1.9.5	Predefined pulsing events . . . . .	29
1.10	Grouping . . . . .	29
1.11	Function definition . . . . .	29
1.12	Files . . . . .	30
1.13	Standard commands, operators or global variables . . . . .	31
1.13.1	<b>echo</b> . . . . .	31
1.13.2	<b>eval</b> . . . . .	31
1.13.3	<b>stop, block, unblock, freeze, unfreeze</b> . . . . .	31
1.13.4	<b>ping</b> . . . . .	31
1.13.5	<b>connections, vars, uservars</b> . . . . .	32
1.13.6	<b>stopall, killall, disconnect</b> . . . . .	32
1.13.7	<b>quit, reboot, shutdown, reset</b> . . . . .	32
1.13.8	Useful functions that are part of the specification . . . . .	32
1.14	URBI Header . . . . .	33

# Introduction

This document is a full specification of the URBI language. It is not a tutorial, it is a reference manual and a technical specification. If you are already familiar with URBI, this document may help you get precise information on a given command or help you review more advanced features not covered in tutorials. If you want an introduction to URBI, try the URBI Tutorial on [www.gostai.com](http://www.gostai.com)

The revision number of this document matches the number of the URBI kernel. So if you want to know what your current version of URBI can do, just get the appropriate specification based upon the kernel number given in the URBI header at start.

# URBI 2.0 Compatibility Program

In preparation for URBI 2.0 currently in development, we have highlighted specific features of URBI 1.0 described here that will probably be modified in future versions. In any case, compatibility will be ensured through conversion scripts and/or backward compatibility within the language, but we advise to try to avoid 1.0 specific constructs whenever possible if compatibility with future versions is a concern.

URBI 2.0 main difference will be the application of the "everything is an object" approach, with extended support for identifier names, allowing constructs such as `f(x) . foo() . bar`, and enhanced support for tags seen as regular objects in the language.



URBI 1.0 only sections are mentioned with the following icon in the margin.

# Chapter 1

## URBI Language Specification

URBI (Universal Real-time Behavior Interface) is a client/server based interpreted language that can be used to control robots or complex systems of any kind. The language defines a standard protocol to give commands and receive messages from the machine to be controlled. The interpreter and the wrapped server are called the "URBI Engine".

We will describe here in extension the features of the URBI language.

### 1.1 URBI Port

By default, the URBI Engine is listening on port 54000 in a TCP/IP connection, unless stated otherwise. In principle, other ways of sending/receiving commands could be available, like using a shared memory between processes when the client and the server are running on the same host or using something else than TCP/IP. This will depend on the underlying hosting OS and would typically be described in the specific "URBI Doc" manual for your application/robot.

### 1.2 Comments

In the following, we will use URBI comments to give extra information about code lines given as examples. URBI comments can be specified like in C, C++ with `'//'` or `'/* ... */'` and also like many scripted languages using `'#'`, which is equivalent to `'//'`.

### 1.3 Commands, Time Operators and Messages

The URBI Engine receives *commands* from a client and returns *messages* to this client. The normal way of using an URBI controlled robot is to send commands using TCP/IP on the URBI port (54000) and wait for messages in return. In a sense, URBI can be seen as a *server of values*.

A simple telnet client is enough to interact with the Engine for simple applications (or the more advanced 'URBI Remote' client), otherwise libraries (liburbi) are available in most programming languages to wrap the TCP/IP sending/receiving job in simple functions.

NB: in URBI 2.0, the concept of "commands" will be uniformly replaced by "expressions", which means that URBI will only see expressions and evaluate expressions. In the following, we keep the term "command", but this term will disappear from 2.0 Specification.

### 1.3.1 Time Operators

In URBI, every command has a duration (beside the execution time required by the processor which is assumed to be negligible from the theory standpoint). Commands are joined by *time operators* like 'semicolon', 'comma', 'pipe' or 'and', which affect the way the commands are serialized or parallelized. We write  $start(A)$  the time when command  $A$  starts and  $end(A)$  the time when it ends. The semantics of the different time operators is the following:

$A;B$

$B$  starts after  $A$  is finished.

$$A;B \iff start(B) \geq end(A)$$

$A,B$

$B$  starts after  $A$  starts.

$$A,B \iff start(B) \geq start(A)$$

$A\&B$

$B$  and  $A$  start at exactly the same time.

$$A\&B \iff start(B) == start(A)$$

**Note on zero time execution commands:**

Zero time execution commands (ZTEC) are commands whose theoretical execution takes zero seconds. Simple assignments, evaluations, etc, are zero time execution commands. Let us consider the three zero time execution commands `zero1`, `zero2` and `zero3`. In the following case, the order of execution of `zero3` compared to `zero1` and `zero2` is undefined:

`{zero1;zero2} & zero3`

might lead to (1): `zero1 zero3 zero2`  
 or: (2): `zero1 zero2 zero3`  
 or: (3): `zero3 zero1 zero2`

The temporal order between `&`-separated commands is undefined by the URBI Specification. In practical situations with multithreaded implementations of the server the set of instructions between brackets will be executed first (option 2), but there is no guarantee.

$A|B$

$B$  starts immediately after  $A$  is finished.

$$A|B \iff start(B) == end(A)$$

### 1.3.2 Tagged commands

Any URBI command can be prefixed by a *tag* followed by a colon.



If no tag is specified, *notag* is assumed to be the default tag.

Examples:

```
x = 12; // notag
my_tag: y = x+12; // the tag is 'my_tag'
```

### 1.3.3 Flags



The tag can be followed by *flags* to ask for meta information/action about the command execution. Any information is given via system messages (prefixed by *\*\*\**). You might use flags without a tag. The available flags are:

- +begin : The system message "*\*\*\* begin*" will be displayed when the command starts.
- +end : The system message "*\*\*\* end*" will be displayed when the command ends.
- +report : This has the combined effect of +begin and +end.
- +error : Notify of any error during the command execution.
- +connection(s) : Run the command in the connection whose ID is given by the string *s*. If *s* is "all", the command will be run in every connections. If *s* is "other", it will be run in every connections except the one from which the command is executed. Beware that no context element from your connection (like variables) are transmitted. You actually run the command in the other connection, not in yours. (WARNING: this flag might completely disappear in future versions, no backward compatibility insured)
- +bg : Run the command in background.

Examples (see below for explanations on system messages):

```
mytag +report: a = 4;
[136901543:mytag] *** begin
[136901543:mytag] *** end
```

### 1.3.4 Grouped commands

Commands can be grouped between brackets like in the following example:

```
{ x=4; y=5, x=x*7 & {y=x-1;w=y+1} | z=y };
```

The group of commands between brackets is considered as a command itself and therefore must be separated from further commands with a time operator (it can also be tagged, stopped, freed or blocked like any other command).



The last command between brackets does not need a time operator.

The bracket-command starts when the first command inside starts and stops when all commands inside have stopped.

## 1.3.5 Messages

When a command returns a value or when it fails, the URBI server returns a message.

The format of a message is the following:

```
[timestamp:tag] message
```

The time stamp is the uptime of the server in milliseconds when the message has been sent. The tag is the tag of the associated command.



If no tag was specified, `notag` is displayed. NB: In URBI 2.0, only the timestamp will be displayed when there is no tag.

The message can be a value (float, string, list or binary) or a system message. System messages can be errors, warnings or information messages, each of them is prefixed by three distinctive characters:

- `!!!` Error messages
- `###` Warning messages
- `***` Information messages

Here is a typical example of commands with their messages in return:

```
1+1;
[136901543:notag] 2.0000000

my_tag:6*6;
[136904711:my_tag] 36.0000000

impossible:1/0;
[136471768:impossible] !!! Division by zero
[136471768:impossible] *** EXPR evaluation failed

+timeout(1s): loop 1,
[145745447:notag] ### Deprecated construct +timeout
```

The ability to tag commands is a key feature of URBI since it allows the client to retrieve the results of a specific command in a flow of messages.

## 1.4 Objects

### 1.4.1 Definition

**WARNING: This description is not complete and do not include the latest object-oriented features relative to objects in URBI 1.0. A more detailed specification will follow URBI 2.0, meanwhile you can refer to the URBI tutorial to know more about current status of object-oriented programming in URBI.**

Every sensor, motor or controllable element of the robot is an *object*, or is also called a *device* when referred to a physical hardware component. It has a name and can be entirely accessed using this name as prefix. With any URBI Engine there should be available an extensive list of the existing devices on that particular robot, in the "URBI Doc" document.

Every device is associated with a set of *attributes* and *methods* that can be accessed using the syntax `device.field` or `device.method(...)`. Attributes are device specific variables, and method are device specific functions.

Examples:

Let us consider the object `headPan` associated with the horizontal motor of the Aibo robot's head. The value of the motor angle can be specified using the `val` attribute:

```
headPan.val = 45;
```

This will set the `headPan` motor to 45 degrees, using the variable `headPan.val`.

Reading the position of the motor is simply done by evaluating the value of the `headPan.val` variable:

```
mytag: headPan.val; // or simply: headPan;  
[178101888:mytag] 45.0178740611
```

## 1.4.2 Complementary fields and methods

Other devices like cameras typically have several fields accessible, like in the case of Aibo (see the URBI Doc of Aibo for more details):

```
camera.resolution = 1;  
camera.format = 0;  
camera.width;  
[188141887:notag] 104.000000
```

In the case of Aibo, the `speaker` device (commanding the robot speaker) has a `play` method that plays a sound given as a wav file on the memorystick:

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

Detailed documentation on what attributes and methods are available on a particular robot device should be available in the robot's "URBI Doc" file, and are generally following the "URBI Ready Certification Standard" to insure compatibility between robots.

## 1.5 Variables

### 1.5.1 Variables types

Variable can have the following types:

- float: expressed with a point
- time: these are in fact floats conversion of expressions like 3h45m12s into milliseconds. Available time symbols are: d, h, m, s, ms. They can be composed at will. The main usage is to facilitate expression of time values in complex commands.
- string: between double quotes and with a C like syntax for quoted symbols
- list: comma separated and between square brackets. Can be heterogeneous, like `[1.2, 3, "hello"]`
- binary
- objects



Note that there is no real integer or boolean type in URBI 1.0. When an integer value is expected, the float is automatically casted into an int.

Binary variables are specified by the keyword `BIN` followed by the size the binary data and a set of freely attributed parameters useful to characterize the binary. A typical use of binary variables is to store sound like wav data, or images as jpeg.



Objects are described by the keyword OBJ followed by a parameterized list of attributes.

In URBI 1.0, objects are not parsable, which means that you cannot do:

```
myobj = OBJ [x:1,y:2]; // does not parse
```

## 1.5.2 Variables syntax



Variable names are composed of a prefix and a suffix separated by a point. When there is no prefix, the variable is assumed to be local to the connection or to the function call (a virtual prefix equal to the connection or function call identifier is silently created). Variable can also be indexed or multi-indexed by integer values (float are rounded to integers) or strings between brackets, allowing to use flexible arrays in URBI.

Examples:

```
i = 4; // local variable. Stored as U123456789.i in memory,
      // U123456789 is the connection identifier.

myprefix.foe = 12; // global variable, visible by any connection

myarray[12] = 4;
mymultiarray[4][17] = 8;
mystringarray["hi"] = 6;
```

Here are a few examples of variables assignments:

```
i = 4;
j = i;
s = "hello";
s2 = s + " world!";
mylist=[1,2,"hello",4];
myemptylist=[];
composelist = [1,2] + [3,4] + "hello";

buffer = bin 10;0123456789 /* buffer contains 10 octets
specified here as the ascii values of 0,1,...,9. */

sound = bin 2048 wav 2 16000 16;
##### 2048 bytes of 16KHz 16 bit stereo wav data #####
```

## 1.5.3 Variables declaration

The declaration of a variable is implicit in URBI and is done via any assignment. Variable used without having been assigned a value first will be rejected as unknown. Variable types are inferred automatically during the first assignment.

### **var, strict, unstrict**

Automatic and implicit variable declaration is a convenient feature for fast scripting but can be the source of bugs for larger programs. The main problem arises from mistyped variable names which can go unnoticed in an assignment.

To provide with a way to allow name-safe coding in URBI, you can explicitly define variables by hand with the `var` keyword and activate the variable definition checking with the command `strict`.

```

a = 0; // that's ok

strict;
b = 0; // b is not implicitly defined
[00025254:notag] *** Unknown identifier: U597753864.b

var b = 0; // now it's ok again

```



Once a variable has been declared (but not assigned), it contains the value `void`. You can check if a variable is `void` with the `isvoid` function:

```

var x;
isvoid(x);
[01311894:notag] 1.000000

```

The syntax of `var` is very flexible. Like in the example above, you can perform an assignment prefixed by `var` to make sure your variable is defined, but you can also define variables independently of assignments, for example at the beginning of your program.

```

var a;
var b;
var stuff.d;

```

The variable definition checking can be turned off with the command `unstrict`. Note that when the definition checking is on, type mismatches are signaled when they occur.

### **delete, isdef**

To release a variable (and the associated memory), the `delete` operator is available:

```

a = 0;
a = "hi";
[00025254:notag] *** U597753864.a type mismatch

delete a;
a = "now it's ok";
showit:a;
[00071210:showit] "now it's ok"

```

The `isdef(x)` function returns 1 when `x` is defined or zero otherwise.

## **1.5.4 Variable properties**

Certain variables are associated to a set of properties. For example, there is a `rangemin` and `rangemax` property available. The property can be read/set via an indirection `->` on the variable name:

```

x = 0;
x->rangemax = 10;
x->rangemax;
[00035124:notag] 10.000000

legRF1.val->rangemin;
[00035834:notag] -134.000000

```

The following properties are available for most numerical attributes (currently, in URBI 1.0 it is available to all variables):

**rangemin**: the minimal value accepted. Outranging values are clipped. default = -inf  
**rangemax**: the maximal value accepted. Outranging values are clipped. default = +inf  
**speedmax**: the maximal speed (units/s) for the evolution of the variable. default = +inf  
**speedmin**: the minimal speed (units/s) for the evolution of the variable in a speed modified assignment. default = 0  
**unit**: variable unit. default = "", usual values include "deg", "cm", "m",...  
**delta**: variable tolerance used with the adaptive modifier (explained in 1.6.1)  
**blend**: variable blend mode (explained in 1.6.5)

### 1.5.5 Variables by name

The syntactic construction  $\$(s)$  where  $s$  is a string is equivalent to the variable whose name is given by  $s$ . For example, the two following constructions are equivalent:

```
 $\$("global.val") <=> global.val$ 
```

Symmetrically, the operator  $\%$  applied on a variable name gives the string equal to the name of the variable:

```
%global.val;  
[01544390:notag] "global.val"
```

Using these features, the name of a variable can be used as a crude mechanism for references when passing function parameters for example. The string type plays the role of a pointer in that case.



The usage of this feature might be deprecated in future versions when actual references will be available, so use with caution.

### 1.5.6 Static variable use



In some cases, it is necessary to have a variable in an expression that will be evaluated only once the first time the expression is calculated and which will keep this original value if the expression is re-evaluated later. This is called "static variable use".

To request a static evaluation of a variable, the name of the variable must be prefixed by the `static` keyword.

The following example shows how the static declaration might affect the behavior of a `wait` command whose test is based on the value of a variable  $x$  (see commands description for the semantics of "wait"):

```
x=0;  
{waituntil (x==1) | ping},  
x=1;  
[00943009:notag] *** pong time=943010.110000  
x=0;  
{waituntil (static x==1) | ping},  
x=1;  
// nothing happens, the value of x is still 0 in the test evaluation.
```

This feature is useful with loop indexes used in modifiers (see below). Since the index will change while the loop is executed, it is necessary that the expression evaluation keeps the original value of the index instead of using the current value (which at the end of the loop will be identical for all instances). Example:

```
for & (i=0; i<5 ;i++) {
  x = 0 time: (100 * static i) |
  echo "timer is over"
}
```



Variables inside array indexes are automatically made static for convenience, since there would be no point doing otherwise. So, `tab[i]` and `tab[static i]` are identical.

### 1.5.7 Variable aliases



The `alias` command can be used to create variable aliases. It can be used for example to rename variables for compatibility or to conveniently access a set of variables using an array and a numerical index.

The `alias` command without parameters shows the list of existing aliases.

When they are used, aliases are chained until they reach a variable for which no alias exists.

```
alias here.a there.b;
alias here.tab[1] there.b1;
alias here.tab[2] there.b2;
alias headPan headPan.val;
alias;
[00943009:notag] *** here.a -> there.b
[00943009:notag] *** here.tab__1 -> there.b1
[00943009:notag] *** here.tab__2 -> there.b2
```



The second parameter of an `alias` command is chained, in case there is already an alias defined for it, but the first parameter is not.

Typical usage of aliases is to define a shortcut to the `.val` field of devices.

### 1.5.8 Variables inside functions

Variables without a prefix inside a function are local to the function call. This is essential for recursive calls.



You can access local variables belonging to your connection from inside a function by using the prefix `local` which will be transformed into the connection prefix.

Consider the following example:

```
i = 4; // local variable. Stored as U123456789.i in memory
function f() {
  i = 7; // i is local to the function call, it is not the same
        // as the one outside
  i;     // returns 7
  local.i; // returns 4
};
```

If the function is an object method, the object attributes in the function are not interpreted as local variable but as the object attributes, unless there is already a local variable defined with the same name (locals override object scope):

```

class myobj {
  var x;
};

myobj.x = 4;
x=6;

function myobj.f() {
  var x; // defines x as a variable local to the function call
  x = 7; // x is local to the function call, it is not the same
        // as the one of object 'obj'
  x;      // return 7
  local.x; // return 6
  self.x; // return 4
};

```

Function arguments are local to the function call.

## 1.5.9 Variable extensions

Variables names can be followed by several "extensions" which will modify the way the variable evaluation or assignment is interpreted. Extensions are defined by an apostrophe, followed by one or several characters describing the extension:

### 'n : normalization

Using the `rangemin` and `rangemax` properties, it is possible to calculate a normalized value between 0 and 1 for any variable. This is done with the 'n extension. This can be useful to achieve a limited sort of robot independent commands. Example:

```

x = 4;
x->rangemin = 0;
x->rangemax = 20;
x;
[00943009:notag] 4.000000
x'n;
[00943009:notag] 0.200000
x'n = 0.5;
x;
[00943009:notag] 10.000000

```

If `rangemin` or `rangemax` are not defined, URBI is unable to calculate the normalized value and complains about it:

```

y = 4;
y'n;
[00943009:notag] !!! Impossible to normalize: no range defined for variable...
[00943009:notag] !!! EXPR evaluation failed

```

### ' : first order theoretical derivative

This extension returns the first order derivative of the variable assignments. When used on a motor device value for example, it returns the theoretical derivative of the input command on the motor (not the actual real motor position derivative as it could be estimated from the sensor output).

```

x = 0;
x->blend = add; // add mode to aggregate assignments

x = 100 speed:33 &
x = 400 speed:12, // the derivative will be the added speeds
x';
[00143009:notag] 45.000000

```

#### **"** : second order theoretical derivative

This extension returns the second order derivative of the variable assignments. Like the first order derivative above, it reflects the theoretical assignments and not the real variable change in case of a motor value.

```

x = 0;
x->blend = add; // add mode to aggregate assignments

x = 100 accel:3 &
x = 400 accel:2.5, // the derivative will be the added speeds
x'';
[00143009:notag] 5.500000

```

#### **'d** : first order real derivative

This extension returns the first order derivative of the variable value. In most cases, it is exactly the same as the theoretical first order derivative described above. In the case of motor values or sensors however, the theoretical value (coming from an assignment command) might differ from the real value taken by the motor and reported by the sensor. This extension measure this real value derivative.

```

motoroff;
// show 10s of the headPan motor value "sensed" derivative
timeout(10s) loop headPan'd;

```

#### **'dd** : second order real derivative

This extension returns the second order derivative of the variable value and is described like the first order real derivative described above.

#### **'e** : "theoretical vs real" error estimation

This extension returns the difference between the theoretical value that the variable should have according to its underlying assignments and the real value coming from sensors. If the variable is not a motor or a sensor, this error is always zero. Otherwise, in the case of motors, it measures the latency in position which often reflects the force currently applied on the motor (the greater the difference, the greater the "effort" to compensate it).

In practical situations, this quantity can be used to measure the force or torque on a motor or to detect potential jams (if the motor is stopped in the course of a trajectory, the error absolute value will grow).

## 1.6 Assignments, Modifiers and blending modes

Assignments in URBI are of the form:

```
variable = value [modifiers]
```

Modifiers are optional parameters that affect the semantics of the assignment. They are described as a list of couples `modifier_name:value` after the main assignment value. *The order of the modifiers in the list has no influence.*

NB: in URBI 2.0, modifiers will be "syntactic sugar" for a more conventional object based syntax. The simplified and easy to use version described here will be maintained however. There are four types of assignments, depending on the type of the value:

### 1.6.1 Float assignment

We describe here the available modifiers for a float assignment. Modifiers for float assignments allows the float value to be assigned with different speed, time limit, motion profile. To use a modifier, the variable that is assigned must have an *initial value*, otherwise a "no start value" error is generated (except for the `sin` modifier). The value to be assigned is called the *target value*.

It is possible to limit the speed at which values are reached using the `speedmin` and `speedmax` properties. Setting `speedmax` to  $s$  (units/seconds) insures that the corresponding variable will never move faster than  $s$ . `speedmin` is taken into account by certain commands only (see below), to insure that the device will move of a minimal significant amount at each time step.

```
time:t
```

`time:t` means that the target value must be reached in  $t$  milliseconds, starting from the initial value ( $t$  can be expressed as a time value, like `3m21s15ms`). A linear trajectory over time is computed and executed. The assignment command terminates when the time is over. The trajectory is not recalculated if the value deviate from the initial trajectory planned. URBI tries to stick back to the pre-calculated trajectory when it can and if there is enough time left.

Example:

```
x = 0; x = 100 time:1s; // reaches value 100 in 1 second.
```

Note that there is no guarantee that the target value will be reached. It is reached only if the time is enough for the system to reach it, taken into account the fact that the speed of some variables linked to motors is limited by the hardware. It will also be reached if there is no other assignment command acting on the corresponding variable just after this command.

```
time:t adaptive:1
```

`time:t` means that the target value must be reached in  $t$  milliseconds, starting from the initial value. The *adaptive* modifier set to 1 indicates that the trajectory should not be pre-calculated but adjusted at each time step. So if the value deviates from the normal trajectory, the command will adapt itself. The command terminates when the target value is actually reached.

Example:

```
x = 0;
x = 100 time:1000 adaptive:1; // reaches 100 in 1 second,
                             // in adaptive mode.
```

If this command is used with a motor, it is possible to set a minimum speed, using the `speedmin` field of the corresponding device. The reason of this feature is that when the speed is too low, the joint will never move and, since the trajectory is recalculated at each time step, it will not evolve properly (the initial value will remain always the same).

The `delta` property of the variable is used to set a tolerance in the definition of "the target value is reached". To be precise, the target value is reached  $\pm$  `variable->delta`. The `delta` property should reflect the hardware precision. Setting `delta` too low could result in the target never being reached because the reaching cannot be detected. Theoretical variables (not related to a device) have a value of zero for `delta` by default.

`smooth:t`

`smooth:t` means that the target value must be reached smoothly in  $t$  milliseconds, starting from the initial value. A smooth trajectory with a sinusoidal profile is computed and executed. The assignment command terminates when the time is over. The trajectory is not recalculated if the value deviate from the initial trajectory planned. URBI tries to stick back to the pre-calculated trajectory when it can and if there is enough time left.

Example:

```
x = 0;
x = 100 smooth:1000; // reaches 100 in 1 second
                    // with a smooth motion profile
```

Note that there is no guarantee that the target value will be reached. It is reached only if the time is enough for the system to reach it, taken into account the fact that the speed of some variables linked to motors is limited by the hardware. It will also be reached if there is no other assignment command acting on the corresponding variable just after this command.

`speed:s`

`speed:s` means that the target value must be reached with a constant speed of  $s$  units per seconds. A linear trajectory over time is computed and executed. The assignment command terminates when the corresponding time, calculated from the speed, is over. The trajectory is not recalculated if the value deviate from the initial trajectory planned. URBI tries to stick back to the pre-calculated trajectory when it can and if there is enough time left.

Example:

```
x = 0;
x = 100 speed:10; // reaches 100 with a speed of 10 units/sec
```

Note that there is no guarantee that the target value will be reached. It is reached only if the time is enough for the system to reach it, taken into account the fact that the speed of some variables linked to motors is limited by the hardware. It will also be reached if there is no other assignment command acting on the corresponding variable just after this command.

```
speed:s adaptive:1
```

`speed:s` means that the target value must be reached with a constant speed of  $s$  units per seconds. The *adaptive* modifier set to 1 indicates that the trajectory should not be pre-calculated but adjusted at each time step. So if the value deviates from the normal trajectory, the command will adapt itself. The command terminates when the target value is actually reached.

Example:

```
x = 0;
x = 100 speed:10 adaptive:1; // reaches 100 with speed 10 units/sec
                             // and does it in adaptive mode
```

If this command is used with a motor, it is possible to set a minimum speed, using the `speedmin` field of the corresponding device. The reason of this feature is that when the speed is too low, the joint will never move and, since the trajectory is recalculated at each time step, it will not evolve properly and might even not move at all (the initial value will remain always the same).

The `delta` property of the variable is used to set a tolerance in the definition of "the target value is reached". To be precise, the target value is reached  $\pm$  `variable->delta`. The `delta` property should reflect the hardware precision. Setting `delta` too low could result in the target never being reached because the reaching cannot be detected. Theoretical variables (not related to a device) have a value of zero for `delta` by default.

```
accel:a
```

`accel:a` means that the target value must be reached with a constant acceleration of  $a$  units per seconds, with an initial speed of zero. A parabolic trajectory is computed and executed. The assignment command terminates when the time corresponding to the execution of the trajectory is over. The trajectory is not recalculated if the value deviate from the initial trajectory planned. URBI tries to stick back to the pre-calculated trajectory when it can and if there is enough time left.

Example:

```
x = 0;
x = 100 accel:10; // reaches 100 with a acceleration of
                  // 10 units/sec2.
```

Note that there is no guarantee that the target value will be reached. It is reached only if the time is enough for the system to reach it, taken into account the fact that the speed of some variables linked to motors is limited by the hardware. It will also be reached if there is no other assignment command acting on the corresponding variable just after this command.

```
sin:t ampli:a phase:p getphase:q
```

`ampli`, `getphase` and `phase` are optional modifiers to be used with the `sin` modifier. Default values are zero for `ampli` and `phase`.

`sin:t ampli:a phase:p getphase:q` means that the value will oscillate in a sinusoidal manner around the target value, with a period of  $t$  milliseconds, an amplitude of  $a$  units and a phase of  $p$  radians. The current value of the phase will constantly be stored in the  $q$  variable ( $q$  must be a valid variable name). The modifier `cos` can be used instead

of `sin` to set the default phase to  $\pi/2$ . The corresponding trajectory is executed in loop and *the command never terminates*.

The trajectory is not recalculated if the value deviate from the initial trajectory planned. URBI always tries to stick back to the pre-calculated trajectory.

Example:

```
headPan'n = 0.5 sin:2s ampli:0.5,  
/* since the command does not terminate, it is better  
   to end it with a comma. This command will oscillate  
   the head will full amplitude in a 2s periodic movement.  
*/
```

Note that by using a conjunction of `phase` and `getphase` on the same variable, you can store the value of the phase on exit and restart where the command was if the assignment is rerun:

```
headPan'n = 0.5 sin:2s ampli:0.5 phase:a getphase:a,
```

```
timeout:t
```

`timeout:t` means that the assignment command will terminate after  $t$  milliseconds, regardless of the motion profile set by other modifiers. This can be especially useful to insure a command will terminate or in conjunction with a `sin` modifier to implicitly set a limited number of oscillation periods.

It is recommended to use a `timeout` command instead. This modifier is kept for compatibility with previous versions.

## 1.6.2 String assignment

Strings can be assigned like floats, and they are delimited by quotation marks.

Example:

```
s = "hello";
```

Modifiers can be used to set local variables used with a dollar prefix inside the string, and compose elaborated strings, a bit like `printf`. The following example illustrates the syntax:

```
s = "hello $name. $name's age is $age" name:"John" age:34;  
s;  
[01544390:notag] "hello John. John's age is 34.000000"
```

## 1.6.3 List assignment

Lists can be assigned like floats or string, and they are delimited by square brackets.

Example:

```
l = [11, "hello", 7.1];  
q = [];  
m = [1,2] + [3,5];  
m;  
[00142582:notag] [1.000000,2.000000,3.000000,5.000000]
```

You can add lists together or with new members. Lists can be heterogeneous. See the `foreach` command to learn how to scan lists. You can access list items directly with an index like `mylist [i] [j]`, get the head or tail:

```

l = [11, "hello", 7.1];
head(l);
[02699247:notag] 11.000000
tail(l);
[02699666:notag] ["hello",7.100000]

```

Lists cannot contain objects in URBI 1.0.

### 1.6.4 Binary assignment

A binary assignment is a very special command in URBI (it has a specific definition in the grammar of the language). It can not be inserted in a flow of commands between brackets or separated by a "comma", a "pipe" or a "and". It must be a single command, terminated by a semicolon. The syntax is:

```
variable = bin <size> <list of parameters>;
```

The size is the number of bytes in the binary variable. The list of parameters is a freely attributed list of float values or identifiers, describing meta information about the binary buffer. This is used for example to specify the sound format in a wav binary. The convention is that the first of these parameters is the format of the binary data, like for example `jpeg`, `raw`, `wav`, `mp3`, `YCbCr`, ...

Just after the ending semicolon comes the binary information. After the specified number of bytes, the URBI parser switch back to ascii mode and waits for another command. The switching between ascii and binary is the reason why binary assignments are special commands handled differently by the parser.

This is an example that plays a sound with Aibo (the server automatically converts the sound format, using the parameters):

```

speaker = bin 2048 wav 2 16000 16 1;
#####
##### 2048 bytes of raw wav data #####
#####...

```

### Hard copies



It is also possible to assign a preexisting binary variable to another variable, like in `newbin = oldbin;`. However, the copy is a pointer copy, to save memory. So, in the given example above, everything that is done to `oldbin` is done to `newbin` too. To make a hard copy of a binary variable, use the `copy` operator:

```
mybin = copy micro; // hard copy of the micro device in Aibo
```

### 1.6.5 Blending modes

Blending modes is an important feature of URBI. Since the server is a multiclient server and the language is parallel, it is possible that two clients (or even the same client) start conflicting assignment commands for the same variable at the same time. To handle this problem, URBI defines six blending modes:

`normal` : This is the default mode for a new variable. When several conflicting assignments run at the same time, the last one to execute sets the value. The others run silently in the background, eventually coming back in front if the last terminates.

- `mix` : In this mode, conflicting commands are mixed and averaged. If one command is to increase with a speed of 10 and the other command is to increase with a speed of 4, the result will be an increase of speed 7.
- `add` : In this mode, conflicting commands are added. If one command is to increase with a speed of 10 and the other command is to increase with a speed of 3, the result will be an increase of speed 13.
- `queue` : Conflicting commands are queued and executed one after the other.
- `discard` : Any conflicting command is ignored and suppressed.
- `cancel` : Any new command that is conflicting with previously existing and running commands will terminate these commands and take the place.

The default mode is `normal`, with one exception: sound binary variables (like the `Aibo speaker.val`) should be generally set to the `queue` mode by default. The effect will be to queue the sounds in the internal sound buffer of the robot. For those sound binary variables, `mix` or `add` acts like a sound mixer. The other modes have an obvious meaning. Blending modes on variables can be checked by evaluation the `blend` property.



It returns a string corresponding to the current blending mode of the variable:

```
foe->blend;
[01544390:notag] "normal"

speaker.val->blend; // or simply: speaker->blend;
[01544390:notag] "queue"
```

Blend mode constants are defined in URBI to set the blend mode easily. These constants are: `add`, `mix`, `queue`, `discard`, `cancel`, `normal`. They are simple shortcuts to the corresponding string (or symbols in URBI 2.0).

```
foe->blend = cancel;
speaker->blend = mix;
foe->blend = "add"; // works also with the explicit string
```

## 1.7 Expressions

The different existing types (float, string, list or binary) can be combined with different operators.

### 1.7.1 Float

The standard `+`, `-`, `/`, `*`, `^` operators are available. The underlying type is the C++ "double".

### 1.7.2 String

For string values, the `+` operator is available for concatenation. Adding a float `f` to a string `s` produces a string composed of the concatenation of the float value in decimal notation and the string `s`:

```
"hello" + " " + "world!";
[01544390:notag] "hello world!"
```

```
"number : "+6;
[01544390:notag] "number: 6.000000"
```

```
"number : "+string(6);
[01544390:notag] "number: 6"
```

The string function is explained below.

### 1.7.3 List

The standard `+` operator is available. It will concatenate two lists: `l1 + l2` will create a list by joining the elements of `l1` and `l2`.

### 1.7.4 Binary

Binary can be concatenated using the `+` operator. The result is a binary containing the concatenation of both buffers. The parameter list is the one of the first binary if it has a list, the second otherwise. This functionality is mainly available to enable the aggregation of sound buffers.



This example uses the `loop` control structure that we will describe later (it is equivalent to `while(true)`), and the `micro` device, available for example on the Aibo robot. It stores 10 seconds of sound in a buffer, using the `+` operator:

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

### 1.7.5 Boolean

Booleans can be used in tests for commands like `if`, `while`, or others. Standard operators like `&&` `||` `!` are available. Tests can be enclosed between parenthesis `( )`. The `true` and `false` constants are available.

Expressions can be compared using the usual `==` `!=` `>` `<` `>=` `<=` comparison operators.

URBI defines also a fuzzy equality test:

`a =~= b` : Means that  $|a - b| \leq a - \delta + b - \delta$ . This fuzzy test uses the `delta` property of the expressions to compute the tolerance on the test. It is extremely useful to test motor values or any type of variable with a significant variance. In future versions of URBI, the delta on both sides will be calculated based on the delta of each element of the expression.

### 1.7.6 Standard functions

There is a set of standard functions, with their usual semantics:

```
sin asin cos acos tan atan exp log round random trunc sqr
sqrt abs
```



There is also a `string` function, which takes a float and returns the integer part of the float converted into a string. This can be useful because the default conversion of a float keeps the decimal values, even if they are zeros.

Examples:

```
sin(pi/2);  
[01544390:notag] 1.000000
```

```
random(100); // random integer value between 0 and 99  
[01544390:notag] 33.000000
```

Here are some useful functions that can be used with strings:

- `strlen(s)` : returns the length of the string *s*.
- `strsub(s, pos, n)` : returns the substring of *s* starting at position *pos* and containing *n* characters. Any overflow is truncated appropriately.

## 1.8 Control and event catching structures

The following control and event catching structures are available in URBI. Some are standard structures common in many known languages like C/C++, some are more specific to URBI. In the examples given, `command1` or `command2` designates a single command or a set of commands enclosed between brackets { }.



The notion of *cycle* is used in the following descriptions. The URBI server is processing the commands at specified intervals (32ms for Aibo robots). The complete examination of the command set is called a "cycle". At every cycle, a `system.update` event is emitted (see event emission in 1.9). The command `noop`, for example, does nothing but takes exactly one cycle to complete, which is equivalent to `wait(system.update)`. This tool is useful to describe the semantic of the following commands.

### 1.8.1 Control structures

```
if (test) command1 else command2
```

This performs a standard `if` branching. Note that there is no terminating symbol between `command1` and `else`. The *else* part is optional.

This command terminates when the selected command (1 or 2) terminates.

Example (see 1.13 for details about the `echo` command):

```
i=4;  
if ( i > 3 ) echo "great" else echo "small";  
[01544390:notag] *** great
```

```
while (test) command
```

This performs a standard `while` loop.

The loop terminates when `command` terminates and the test evaluation returns false.



For the moment, there is at least one cycle between each execution of `command` (technically speaking: `command` is executed in parallel (&) with `noop`). So, it is possible to create an infinite loop but the server will never freeze. \*This will change in future specifications and the `noop` inclusion might disappear\*

Example:

```
i=0;
while (i<=2) {
    i:echo i;
    i++
};
[09696528:i] *** 0
[09696558:i] *** 1
[09696590:i] *** 2
```

Note how the time stamp is incremented at each turn.

```
while | (test) command
```

This is similar to the `while` command except that each instance of `command` is connected with a `|` time operator. This is equivalent to:

```
command_instance1 | // perform test
command_instance2 | // perform test
...
```

The loop terminates when the last instance of `command` terminates and the test evaluation returns false.

Note that in the case of `while |`, it is possible to create an infinite loop that would freeze the server.

Example:

```
i=0;
while | (i<=2) {
    i:echo i;
    i++
};
[09696528:i] *** 0
[09696528:i] *** 1
[09696528:i] *** 2
```



Note how the time stamp is identical at each turn. The `while |` is almost the same than the `while` command except that since the `|` insures that all commands will be executed without a time gap, there is no cycle loop lost in the execution process. It can be useful to speed up the execution of a code, but it might also freeze the server.

```
for (instruction1; test; instruction2) command
```

This has the usual semantics of the C "for". `instruction1` and `instruction2` must be single instructions (no time operator or brackets). It is equivalent to:

```
instruction1;
while (test) {
    command |
    instruction2
}
```

```
for | (instruction1; test; instruction2) command
```

Just like the `for` command above, this is equivalent to:

```

instruction1;
while | (test) {
    command |
    instruction2
}

```

**foreach** variable **in** list { command }

This is similar to the for command except that the iteration is done on a list with an explicit variable given to contain the element of the list currently iterated. Also, note that the opening and closing brackets are necessary, even if the command is a single instruction.

NB: If the list is modified during the execution of the foreach command, it will not impact the execution as it was planned at the beginning of the foreach evaluation.

Example:

```

foreach element in [1,2,3] {
    element:echo element;
}
[09696528:element] *** 0
[09696528:element] *** 1
[09696528:element] *** 2

```

**foreach &** variable **in** list { command }

This is similar to the foreach command except that the iteration is done in parallel on each element of the list.

Example:

```

foreach& element in [1,2,3] {
    element:echo element;
}
[09696528:element] *** 0
[09696528:element] *** 1
[09696528:element] *** 2

```

**foreach |** variable **in** list { command }

This is similar to the foreach command except that each instance of command is connected with a | time operator.

Example:

```

foreach| element in [1,2,3] {
    element:echo element;
}
[09696528:element] *** 0
[09696528:element] *** 1
[09696528:element] *** 2

```

**loop** command

loop is equivalent to while (true). It loops and never terminates.

```
loopn (num) command
```

`loopn` (num) loops *num* times the command and terminates. It is equivalent to (but *n* is virtual here):

```
foreach n in list_of_size_num
{ command }
```

```
loopn & (num) command
```

`loopn &` (num) loops *num* times the command in parallel, with the time operator `&`, and then terminates.

It is equivalent to (but *n* is virtual here):

```
foreach& n in list_of_size_num
{ command }
```

```
loopn | (num) command
```

`loopn |` (num) loops *num* times the command in series, with the time operator `|`, and then terminates.

It is equivalent to (but *n* is virtual here):

```
foreach| n in list_of_size_num
{ command }
```

## 1.8.2 Soft tests

Some event catching structures like `at`, `whenever` or `wait` can make use of *soft tests*. Those tests are called *soft tests* because they can integrate an hysteresis threshold, prefixed by the `~` separator. Consider the two following example of soft test:

```
( headSensor == 1 ~ 50ms )
~ 50ms indicates that the test must be true during a least 50ms before the soft
test turns to true.
```

These "soft tests" are very useful in applications like robotics to conveniently set conditions that are fuzzy and more resistant to noise, and to avoid constant triggering on and off around the target value of the test.

## 1.8.3 Event catching structures

```
at (softtest) command1 onleave command2
```

The `at` command is an event catcher. This command never terminates and run in the background. At the moment when the test becomes true, `command1` is executed once. At the moment when the test becomes false again, `command2` is executed once, if it is defined (the `onleave` part is optional). Then again, `at` waits for the test to become true to execute `command1` once and so on.

The difference between `at` and `if` is that `if` does the test only once. If the test fails, it will never try again because the `if` command terminates. The `at` command never terminates

and performs the test at each cycle. It will also require that the test becomes false before the command can be executed again.

```
whenever (softtest) command1 else command2
```

The `whenever` command is also an event catcher. This command never terminates and runs in the background. Whenever the `softtest` is true, `command1` is executed, otherwise, if it is given, `command2` is executed (the `else` part is optional). `command1` or `command2` must terminate before the soft test is reexamined again. So if one of those commands does not terminate, the event catching capability will be lost.

`whenever` is different from `at` because `command1` will be executed each time the test is true, and not only at the time it *becomes* true. In that sense, `whenever` is close to `while`, the difference being that when the test fails it does not terminate but waits for the test to become true again.

```
wait n
```

`wait n` waits  $n$  milliseconds and terminates. This is useful to make a pause or control the execution time or synchronization of different commands.

```
waituntil (softtest)
```

`waituntil (softtest)` waits until `softtest` becomes true and terminates. This is useful to make a pause or control the execution time or synchronization of different commands.

```
timeout (time) command
```

`timeout (time) command` runs `command` and stops it when the specified `time` is over. If the command ends before the specified time, `timeout` has no effect. `time` is expressed in milliseconds or with a time expression like `12m54s`.

```
stopif (softtest) command
```

`stopif (softtest) command` runs `command` and stops it if the specified `softtest` becomes true. If the command ends before the specified `softtest` is true, `stopif` has no effect.

```
freezeif (softtest) command
```

`freezeif (softtest) command` runs `command` and freezes it if the specified `softtest` becomes true. When it turns false again, the command is unfrozen. This behavior ends when the command terminates.

## 1.9 Emitting events

Event programming is a fundamental feature and a good way of writing asynchronous programs. The basic idea of event programming is that some command emits an event and another catches it and does something with it.

### 1.9.1 Simple events

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

```
at (myevent) echo "event triggered!";
emit myevent;
[00143009:notag] *** event triggered!
```



Note that the `myevent` 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.myevent`.

### 1.9.2 Events with parameters

You can add parameters to events like this:

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

The parameters can be retrieved when the event is caught:

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

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

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

```
emit myevent(1, "hello");
[00143009:notag] *** catch two: 1.000000 hello
[00143009:notag] *** catch one: hello
emit myevent(2,15);
[00143009:notag] *** catch two: 2.000000 15.000000
```



Note that the variables used to catch the event parameters are local variable and do not have a scope specific to the `'at'` command in URBI 1.0 (this will change in URBI 2.0). To enforce a scope local to the event catching code, you can use a function call that will transmit the parameters with a proper scope local to the function call:

```
at (myevent(x,y)) mycallback_function(x,y);
```

### 1.9.3 Event duration

Simple events 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 it between parenthesis like this:

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

This will induce a difference between `at` and `whenever` event catcher.

### 1.9.4 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 1.13.3):

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

## 1.9.5 Predefined pulsing events

If the URBI server is implemented in a cycle, like on the Aibo, the global event `system.update` pulses at every loop. You can use it as a heartbeat for your programs.

## 1.10 Grouping

An important feature of URBI is the capacity to group objects. This is done with the `group` command:

```
group groupname { name1, name2, ... }
```

Example:

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

This grouping feature is useful to call a *method* on an group, or to make a *multi-device assignment* (broadcasting): the command is passed recursively to any child subgroup and executed to the final object group members. In other terms, using the example above, the command `legLFs.val = 0` will set the value of `legLF1`, `legLF2` and `legLF3` to 0.

Example:

```
group ab {a,b};
ab.n = 5;
a:a.n, b:b.n;
[09696528:a] 5.000000
[09696528:b] 5.000000
```

Object grouping will also affect the way events are processed. If `a` is a group containing `b1` and `b2`, every `a.x` event will trigger an `b1.x` and `b2.x` event.

## 1.11 Function definition



Functions can be defined in URBI using the `function` 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).

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, with the exception of attributes of objects in a method definition. If local function arguments or variables exist with the same name as the object attribute, the local definition overrides the object attribute.

Example:

```

function add(x,y)
{
  z = x + y;
  return z
};

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

```

Recursive function calls are allowed in URBI, since all local variables and parameters are local to the function call.

## 1.12 Files



If the robot has a file system available, it is possible to do some very simple file manipulation. NB: In URBI 2.0, this will be handled by a file I/O library. The following does not necessary apply to this future version:

```
load(file)
```

This command loads the file with name *file* given as a string (be careful not to use spaces or accents in the name), parse its content and execute the commands, as if they had been typed by hand at the position of the "load".

```
save(file,s)
```

This command creates a file with name *file* (*file* is a string) and stores the string *s* in it. The file can be reloaded later with the `load` command.

```
URBI.INI
```

The URBI.INI file, located at the root of the file system, is executed when the server starts. It is typically used to execute some initialization commands specific to the robot, like playing a welcome sound.

The commands are executed in a virtual connection called the "ghost connection". Any output from this connection is sent to the debug output, which is most of the case port 59000.

A common usage is to load some custom files at the beginning of the URBI.INI file, to activate "modules". The most usual module is "std.u" which contain standard groups and functions useful for your robot.

```
CLIENT.INI
```

The CLIENT.INI file, located at the root of the file system, is executed when a new client starts. It runs in the new client's connection and any output will be displayed to this connection.

## 1.13 Standard commands, operators or global variables

### 1.13.1 echo

The `echo` command displays a value in a system message. Used with a binary value, it only displays the header containing the bin size and the parameters. This is useful to control the content of a binary without viewing the binary buffer itself.

Examples:

```
echo 45;  
[01544390:notag] *** 45
```

```
echo "hello";  
[01544390:notag] *** hello
```

```
echo camera;  
[01544390:notag] *** BIN 9437 jpeg 208 160
```

### 1.13.2 eval

`eval (s)` parses the string *s* and run the corresponding command, as if it had been typed by hand.

Examples:

```
eval( "a=1;" );  
a;  
[01544390:notag] 1.000000
```

### 1.13.3 stop, block, unblock, freeze, unfreeze

These are fundamental and very useful commands to control the execution of programs:

`stop thetag` will stop any command whose tag is *thetag*.

`block thetag` will block any command whose tag is *thetag*, preventing it to be executed.

`unblock thetag` will unblock the commands whose tag is *thetag*, previously blocked by a `block` command.

`freeze thetag` will freeze any command whose tag is *thetag*, putting it in a "suspended" mode

`unfreeze thetag` will unfreeze the commands whose tag is *thetag*, resuming it where it was when it was frozen.

### 1.13.4 ping

The `ping` command sends the following system message:

```
*** pong time=t
```

*t* is the server uptime in milliseconds, with microseconds expressed as decimals. This is the real server time at the moment when the message is sent, it may and will differ from the time stamp in the message header and can be used to do some accurate time measures on the server.

### 1.13.5 connections, vars, uservars



`connections` lists the currently opened connections. The "Ghost Connection" is the connection running `URBI.INI`.  
`vars` show all variables.  
`uservars` show all user defined variables.

### 1.13.6 stopall, killall, disconnect



The commands `killall` and `disconnect` are followed by a connection identifier. Each client has a unique connection identifier, displayed at start in the URBI header and accessible later through the `connectionID` variable.  
`killall id` will empty the command tree of the connection whose ID is *id*. Every command will be stopped (WARNING: this function might change in future versions).  
`disconnect id` will close the connection whose ID is *id* (WARNING: this function might change in future versions).  
`stopall` is equivalent to `killall` on every connection opened. This is a powerful command that stops everything running on the robot and should be used carefully.

### 1.13.7 quit, reboot, shutdown, reset

`quit` closes the current connection (WARNING: this function might change in future versions)

`reboot` reboots the robot and `shutdown` switches the robot off.

`reset` reset the robot by doing a "software" reboot: all functions are deleted, all variables are lost and `URBI.INI` and `CLIENT.INI` are reloaded.

### 1.13.8 Useful functions that are part of the specification



The following functions are part of the URBI specification:

- `freemem()` : returns the amount of free memory in bytes.
- `power()` : returns the remaining battery power, expressed as a float between 0 and 1 (0:empty, 1:full).
- `time()` : returns the server uptime, in milliseconds, with a microsecond precision in the decimals if it is available for the robot.
- `cpuload()` : returns the load of the robot CPU: 0 for idle and 1 for full capacity. Going above 1 will degrade the quality of the executed commands and time constraints are not guaranteed anymore.
- `loadwav("file.wav")` : this will return a binary equal to the wav file specified. Useful to store in the URBI memory some frequently used sound files.

## 1.14 URBI Header

When a new client starts, it must receive a standard URBI header for the server to be URBI compliant.



Here is an example of a URBI Header coming from the 1.0 version of URBI server for Aibo robot, as distributed on [urbiforge](http://urbiforge.com):

```
[00020380:start] *** *****
[00020380:start] *** URBI Language specif 1.0 - Copyright (C) 2006 Gostai SAS
[00020380:start] *** URBI Kernel version 1.0 rev. 100
[00020380:start] ***
[00020380:start] ***   URBI Engine 1.0 for Aibo ERS2xx/ERS7 Robots
[00020380:start] ***   (C) 2004-2006 Gostai SAS
[00020380:start] ***
[00020380:start] *** URBI comes with ABSOLUTELY NO WARRANTY;
[00020380:start] *** This software is free, and you are welcome to use
[00020380:start] *** it under certain conditions; see LICENSE for details.
[00020380:start] ***
[00020380:start] *** See http://www.urbiforge.com for news and updates.
[00020380:start] *** *****
[00020380:ident] *** ID: U595075704
```