URBI Language Specification

v 1.0

Jean-Christophe Baillie

January 2005
1 URBI Language Specification

1.1 Comments ............................................ 4
1.2 Commands, Time Operators and Messages ............................................ 4
  1.2.1 Time Operators ....................................... 4
  1.2.2 Tagged commands ...................................... 5
  1.2.3 Flags ............................................ 5
  1.2.4 Grouped commands ..................................... 6
  1.2.5 Messages ........................................... 6
1.3 Devices and variables ....................................... 6
  1.3.1 Variables ........................................... 6
  1.3.2 Devices ............................................ 7
  1.3.3 Device operators ....................................... 8
  1.3.4 Variables by name ...................................... 9
  1.3.5 Static variable use ...................................... 9
  1.3.6 Variable aliases ....................................... 9
1.4 Assignments, Modificators and blending modes ..................................... 10
  1.4.1 Float assignment ....................................... 10
  1.4.2 String assignment ....................................... 13
  1.4.3 Binary assignment ...................................... 13
  1.4.4 Blending modes ....................................... 14
1.5 Expressions ............................................ 14
  1.5.1 Float ............................................ 14
  1.5.2 String ............................................ 15
  1.5.3 Binary ............................................ 15
  1.5.4 Boolean ............................................ 15
1.6 Control and event catching structures ............................................. 16
  1.6.1 Control structures ....................................... 16
  1.6.2 Soft tests ........................................... 19
  1.6.3 Event catching structures .................................. 19
1.7 Grouping ............................................ 20
1.8 Function definition ......................................... 21
1.9 Files ................................................. 21
1.10 Standard commands, operators or global variables .................................. 22
  1.10.1 echo .................................................. 22
  1.10.2 exec .................................................. 22
  1.10.3 stop, block, unblock .................................... 22
  1.10.4 ping .................................................. 23
  1.10.5 motoron, motoroff ....................................... 23
  1.10.6 killall, disconnect, quit .................................. 23
  1.10.7 quit, reboot, shutdown .................................... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10.8</td>
<td>Useful functions that are part of the specification</td>
<td>23</td>
</tr>
<tr>
<td>1.11</td>
<td>URBI Header</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td><strong>URBI grammar</strong></td>
<td>25</td>
</tr>
<tr>
<td>2.1</td>
<td>Bison grammar</td>
<td>25</td>
</tr>
<tr>
<td>2.2</td>
<td>Terminal symbols and flex regexp</td>
<td>28</td>
</tr>
</tbody>
</table>
This document is a full specification of the URBI language. Any implementation of a URBI server must comply with the latest version of this specification to be "URBI compliant". It must also pass through the test softwares that will be based on this specification. See the URBI Language License for more information.

The URBI Language copyright holder is currently the only authority which has the ability to deliver a "URBI compliant" certification.
Chapter 1

URBI Language Specification

URBI (Universal Robotic Body 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.

We will describe here in extension the features of the URBI language that must be implemented in any URBI compliant server.

1.1 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 ‘//’.

1.2 Commands, Time Operators and Messages

The URBI server receives commands from a client and returns messages to this client. The normal way of using a URBI controlled robot is to send commands using TCP/IP on the URBI port (54000) and wait for messages in return. A simple telnet client is enough to do that for simple applications, otherwise libraries (liburbi) are available in most programming languages to wrap the TCP/IP sending/receiving job in simple functions.

1.2.1 Time Operators

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

\[
\begin{align*}
A; B & \quad B \text{ starts after } A \text{ is finished.} \\
A, B & \quad B \text{ starts after } A \text{ starts.}
\end{align*}
\]

\( A; B \iff \text{start}(B) \geq \text{end}(A) \)

\( A, B \iff \text{start}(B) \geq \text{start}(A) \)
\[ A \& B \]

B and A start at exactly the same time.

\[ A \& B \iff \text{start}(B) == \text{start}(A) \]

**Note on zero time execution commands:**

Zero time execution commands 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 \texttt{zero1}, \texttt{zero2} and \texttt{zero3}. In the following case, the order of execution of \texttt{zero3} compared to \texttt{zero1} and \texttt{zero2} is undefined:

\[
\{\texttt{zero1;zero2}\} \& \texttt{zero3}
\]

might lead to (1): \texttt{zero1 zero3 zero2}  
or: (2): \texttt{zero1 zero2 zero3}

The temporal order between \&-separated commands is the same as the order of declaration of the commands but the way they are interleaved is undefined by the URBI Specification. In practical situations, and in the current monothreaded implementations of the server that we know, the set of instructions between brackets will be executed first (option 2), but there is no guarantee.

\[ A \lvert B \]

B starts immediately after A is finished.

\[ A \lvert B \iff \text{start}(B) == \text{end}(A) \]

### 1.2.2 Tagged commands

Any URBI command can be prefixed by a tag followed by a colon. If no tag is specified, \texttt{notag} is assumed to be the default tag.

**Examples:**

\[
x = 12; // \texttt{notag} 
\text{my\_tag}: y = x+12; // \text{the tag is ‘my\_tag’}
\]

### 1.2.3 Flags

The tag can be followed by flags to ask for meta information about the command execution. The information is given as system messages (prefixed by ***). The available flags are:

- \texttt{+begin} : The system message "*** begin" will be displayed when the command starts.
- \texttt{+end} : The system message "*** end" will be displayed when the command starts.
- \texttt{+report} : This has the combined effect of \texttt{+begin} and \texttt{+end}.
- \texttt{+error} : Notify of any delay or error during the command execution. This can be very useful to detect motor jams on the robot, since URBI will detect that the motor movement is not executing properly.

**Examples (see below for explanations on system messages):**

\[
\text{my\_tag} +\text{report}: a = 4; \\
[136901543:my\_tag] *** begin \\
[136901543:my\_tag] *** end
\]
1.2.4 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 end with a time operator (it can also be tagged, 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.

1.2.5 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 or `notag` if there was no tag specified. The message can be a value (float, string or binary) or a system message prefixed by three stars: `***`. Error messages or information messages are system messages.

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

```
1+1;
[136901543:notag] 2.0000000000

my_tag:6*6;
[136904711:my_tag] 36.0000000000

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

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

1.3 Devices and variables

1.3.1 Variables

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 (a virtual prefix equal to the connection identificator is silently created). Variable can also be indexed or multi-indexed by integer values (float are rounded to integers) between brackets, allowing to use arrays in URBI.

Examples:

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

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

myarray[12] = 4;
mymultiarray[4][17] = 8;
```
Note that arrays are internally handled as the variable followed by a double underscore and the index: __index. In the above case, myarray__12 and mymultiarray__4__17.

Variable can be floats, strings (between double quotes) or binaries. Note that there is no integer type in URBI. 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. Here are a few examples of variables assignments:

```urbi
i = 4;
j = i;
s = "hello";
s2 = s + " world!";

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 ######
```

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 and cannot be changed after (type mismatch error). To release a variable (and the associated memory), the `delete` operator is available:

```urbi
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"
```

### 1.3.2 Devices

Every sensor, motor or controllable element of the robot is a *device*. It has a device name and can be entirely accessed using this name as prefix. With any URBI server there should be available an extensive list of the existing devices on that particular robot.

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

Examples:

Let us consider the device *headPan* associated with the horizontal motor of the Aibo robot’s head. The value of the motor angle can be specified using the *val* field:

```urbi
motoron; // should be done at start to activate the motors
headPan.val = 45;
```

This will set the *headPan* motor to 45 degrees. Note that it is necessary to start the motors with `motoron` to be able to use this command. `motoroff` is also available.

Reading the position of the motor is done by evaluating the value of the `headPan.val` variable:
headPan.val;
[178101888:notag] 45.0178740611

In parallel to val, there is also the valn field which can be set or read as a normalized value between 0 and 1 for the device, according to the maximal and minimal range this device can take. This can be useful to achieve a limited kind of robot independent commands. Modifying val also modifies valn and reciprocally.

Other devices like cameras typically have several fields accessible, like in the case of Aibo:

camera.resolution = 1;
camera.format = 0;
camera.width;
[188141887:notag] 104.0000000

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 fields and methods are available on a particular robot device should be available in the robot’s URBI server documentation.

However, some fields are mandatory for any device in a URBI compliant server. The val and valn fields are mandatory. For devices relative to joints or motor, the 'load' field is also mandatory. It controls the load of the joint: 0 means that the joint is loose, 1 means that the joint is not movable by hand (any applied torque will be exactly compensated). Any value in between gives an intermediary situation.

### 1.3.3 Device operators

Some operators are available to access devices information. rangemin and rangemax return the min and max value of the val field of the device (used to interpret valn). info gives detailed information on the device (it returns nothing, the information is given as system messages). unit returns the device unit for the val field, given as a string. blend is a parameterized operator that will be described later. Examples:

```python
the_rangemin: rangemin headPan;
[01376392:the_rangemin] - 91.000000

the_rangemax: rangemax legRF1;
[01376793:the_rangemax] 120.000000

unit legRF1;
[01376998:notag] "deg"

info headPan;
[01544390:notag] *** description: Head pan
[01544390:notag] *** device: headPan
[01544390:notag] *** current value: 73.912655
[01544390:notag] *** current load: 1.000000
[01544390:notag] *** rangemin: -91.000000
[01544390:notag] *** rangemax: 91.000000
[01544390:notag] *** unit: deg
```
1.3.4 Variables by name

The syntaxic 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:

\[
\$("global.val") \leftrightarrow \text{global.val}
\]

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

\[
%\text{global.val};
\]

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

1.3.5 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 a colon.

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"):

\[
\begin{align*}
x &= 0; \\
\{\text{wait (x==1) | ping}, \\
x &= 1; \\
& [00943009:\text{notag}] \text{ *** pong time=943010.110000} \\
x &= 0; \\
\{\text{wait (:x==1) | ping}, \\
x &= 1; \\
& \text{// nothing happens, the value of x is still 0 in the test evaluation.}
\end{align*}
\]

This feature is useful with loop indexes used to make references to arrays in modificators or tests. 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. Example:

\[
\begin{align*}
\text{for & (i=0; i<5 ;i++)} \\
& \text{ { wait timer[:i] |} } \\
& \text{ echo "timer is over" }
\end{align*}
\]

1.3.6 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 access a set of variables using an array.

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.

\[
\begin{align*}
\text{alias here.a there.b;} \\
\text{alias;}
\end{align*}
\]

The second parameter of the alias command is chained, but not the first.
1.4 Assignments, Modificators and blending modes

Assignments in URBI are of the form variable = value [modificators]. Modificators 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 modificators in the list has no influence.

There are three types of assignments, depending on the type of the value:

1.4.1 Float assignment

We describe here the available modificators for a float assignment. Modificators for float assignments allows the float value to be assigned with different speed, time limit, motion profile... To use a modificator, the variable that is assigned must have an initial value, otherwise a "no start value" error is generated. The value to be assigned is called the target value.

For variables related to devices, like `device.val`, it is possible to limit the speed at which values are reached using the speedmin and speedmax fields. Setting speedmax to $s$ (units/milliseconds) insures that the corresponding device 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` means that the target value must be reached in $t$ milliseconds, starting from the initial value. 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:

\[
x = 0;
\]
\[
x = 100 \text{ time:}1000; \quad // \text{reaches 100 in 1 second.}
\]

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` means that the target value must be reached in $t$ milliseconds, starting from the initial value. The adaptive modificator set to 1 indicates that the trajectory should not be precalculated 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:

\[
x = 0;
\]
\[
x = 100 \text{ time:}1000 \text{ adaptive:}1; \quad // \text{reaches 100 in 1 second,}
\]
\[
\quad // \text{in adaptive mode.}
\]

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).
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:

\[
\begin{align*}
x &= 0; \\
x &= 100 \text{ smooth:}1000; \quad \text{// reaches 100 in 1 second} \\
&\quad \text{// with a smooth motion profile}
\end{align*}
\]

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:

\[
\begin{align*}
x &= 0; \\
x &= 100 \text{ speed:}10; \quad \text{// reaches 100 with a speed of 10 units/sec}
\end{align*}
\]

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 modificator set to 1 indicates that the trajectory should not be precalculated 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:

\[
\begin{align*}
x &= 0; \\
x &= 100 \text{ speed:}10 \text{ adaptive:1;} \quad \text{// reaches 100 with speed 10 units/sec} \\
&\quad \text{// and does it in adaptive mode}
\end{align*}
\]
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).

`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:

```plaintext
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`

`ampli` and `phase` are optionnal modificators to be used with the `sin` modificator. Default values are zero for both.

`sin:t ampli:a phase:p` 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 modificator `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:

```plaintext
headPan.valn = 0.5 sin:2000 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. */
```

`timeout:t`

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

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

Example:

\[ s = "hello"; \]

Modificators 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 illustrate the syntax:

\[ s = "hello $name. $name’s age is $age" name:"John" age:34; \]
\[ s; \]
\[ "hello John. John’s age is 34.000000" \]

1.4.3 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:

\[ \text{variable} = \text{bin} \ <\text{size}> \ <\text{list of parameters}>; \]

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.

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

\[ \text{speaker.val} = \text{bin} \ 2048 \ \text{wav} \ 2 \ 16000 \ 16 \ 1; \]
\[ \text{-----------------------------------------------} \]
\[ \text{### 2048 bytes of raw wav data ###} \]
\[ \text{-----------------------------------------------} \]

Hard copies

It is also possible to assign a preexisting binary variable to another variable, like in \texttt{newbin = oldbin;}.

However, the copy is a pointer copy, to save memory. So, in the previous case, everything that is done to \texttt{oldbin} is done to \texttt{newbin} too. To make a hard copy of a binary variable, use the \texttt{copy} operator:

\[ \text{mybin} = \text{copy} \ \text{micro.val}; \ // \text{hard copy of the micro device in Aibo} \]
1.4.4 Blending modes

Blending modes is an important feature of URBI. Since the server is a multiclient server, it is possible that two clients (or even the same client) start conflicting assignement commands for the same variable at the same time. To handle this problem, URBI defines five 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.
- **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 decrease with a speed of 3, the result will be an increase of speed 7.
- **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** acts like a sound mixer. The other modes have an obvious meaning.

Blending modes on variables can be checked using the `blend` operator followed by the variable name. It returns a string corresponding to the current blending mode of the variable:

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

blend speaker.val;
[01544390:notag] "queue"
```

Blending modes can be set using the `blend` operator followed by the blend mode between square brackets (possible values are: **mix**, **queue**, **normal**, **cancel**, **discard**) and the variable name:

```
blend[cancel] foe;
blend[mix] speaker.val;
```

1.5 Expressions

The three existing types (float, string or binary) can be combined with different operators. There is a boolean type available for tests but not for variables.

1.5.1 Float

The standard `+-/*` operators are available. Several useful functions with their usual semantics are also given:

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

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

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

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.

1.5.3 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 aggregate sound buffers.

This example uses the for control structure that we will describe later and the micro device, available for example on the Aibo robot. It stores sound in a buffer, using the + operator:

```urbi
sound = bin 0;
for (i=0;i<100;i++)
    sound = sound + micro.val;
```

1.5.4 Boolean

Booleans are expressions available in tests for commands like if, while, or others. They cannot be stored in a variable but only evaluated in a test. 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 some fuzzy equal tests:
\( a \sim b : \) Means that \( |a - b| \leq \text{global.\epsilon_tilde} \)

\text{global.\epsilon_tilde} is a global variable with default value equal to 1.0. This value can be modified, like for any other variable.

\( a \% b : \) Means that \( |1 - a/b| \leq \text{global.\epsilon_percent} \)

\text{global.\epsilon_percent} is a global variable with default value equal to 0.05. This value can be modified, like for any other variable.

Tests can be evaluated and converted to a number (0 for false and 1 for true) using the \texttt{booleval} operator:

\begin{verbatim}
booleval (4 == 5);
[01544390:notag] 0.000000
\end{verbatim}

### 1.6 Control and event catching structures

The following control and event catching structures are available in URBI. Some are standard structures, some are more specific to URBI. In the examples given, \texttt{command1} or \texttt{command2} designate a single command or a set of commands enclosed between brackets \{ \}.

The notion of \textit{cycle} is used in the following. The URBI server is processing the command tree at specified intervals (32ms for Aibo robots). The complete examination of the command tree is called a cycle. The command \texttt{noop}, for example, does nothing but takes exactly one cycle to complete. This is useful to describe the semantic behavior of the following commands.

In the following, when it is specified that the server might freeze, it will not indeed freeze because of security in URBI that automatically terminates any command that loops more than 10000 turns in one cycle (can be modified with the \texttt{global.maxloop} variable). In real situations, it is not possible to afford to actually freeze the server because of a code bug. And since URBI is a command execution script language, not designed for scientific calculation or other loop hungry applications, the \texttt{global.maxloop} limit is not a problem in most cases.

#### 1.6.1 Control structures

\begin{verbatim}
if (test) command1 else command2
\end{verbatim}

This performs a standard if branching. Note that there is no terminating symbol between \texttt{command1} and \texttt{else}. The \texttt{else} part is optional.

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

Example (see 1.10 for details about the \texttt{echo} command):

\begin{verbatim}
i=4;
if ( i > 3 ) echo "great" else echo "small";
[01544390:notag] *** great
\end{verbatim}

\begin{verbatim}
while (test) command
\end{verbatim}

This performs a standard while loop.

The loop terminates when \texttt{command} terminates and the test evaluation returns false. There is at least one cycle between each execution of \texttt{command} (technically speaking: \texttt{command} is executed in parallel (&) with \texttt{noop}). So, it is possible to create an infinite loop but the server will never freeze.
Example:

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

Note how the timestamp 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:

```c
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:

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

Note how the timestamp 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, their 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:

```c
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
}
```

**for &** (instruction1; test; instruction2) **command**

This is similar to the `for` command except that each instance of `command` is connected with a `&` time operator. More precisely, this is equivalent to:

```
instruction1;
{ command_instance1 & { instruction2 | // perform test
    { command_instance2 & { instruction2 | // perform test
        ... }}}}}
```

The loop terminates when all the instances of `command` and `instruction2` terminate and the test evaluation returns false.

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

Example:

```
for & (i=0;i<=2;i++)
    i:echo i;
[09696528:i] *** 0
[09696528:i] *** 1
[09696528:i] *** 2
```

Note how the timestamp is identical at each turn. The value of `i` changes because the `echo` have virtually an execution time of zero seconds and so `instruction2` (here "i++"), is executed right after the `echo`. This of course is the reason why the loop terminates at some point.

**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):

```
for (n=0; n<num; n++)
    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):
for & (n=0; n<num; n++)
    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):

for | (n=0; n<num; n++)
    command

1.6.2 Soft tests

In the following, 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 examples of soft tests:

( headSensor.val == 1 ∼ 5 )
The ∼ 5 part indicates that the test must be true 5 times before the soft test can be considered as true. It must also be false 5 times before the soft test becomes false.

( headSensor.val == 1 ∼ 50ms )
In that case, ∼ 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 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.

1.6.3 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. Then, 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. command1 and 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.

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.

at & (softtest) command1 onleave command2

The at & command is similar to the at command except that command1 and command2 are executed in parallel with the at command when they are executed. In other words, the softtest is reexamined at each cycle, even if the commands are not terminated yet. For example, there can be several instances of command1 running at the same time if the test switch between false and true several times during the execution time of the first command1.
If one of the commands does not terminate, the event catching capability will not be lost.

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

The whenever command is also an event catcher. This command never terminates and run 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.

```plaintext
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.

```plaintext
wait (softtest)
```

`wait (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.

### 1.7 Grouping

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

```plaintext
group virtualdevice { device1, device2, ...}
```

Example:

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

This grouping feature is useful when a `method` is called on a device (or a virtual device), or to make a `multi-device assignment`: the command is executed for this device and then it is recursively passed to any child subdevice. In other terms, using the example above, the command `legLF.val = 0` will set the value of `legLF1`, `legLF2` and `legLF3` to 0. This child passing mechanism differs from the usual object oriented inheritance mechanism because object oriented hierarchies are based on "is-a-kind-of" relationships whereas the grouping hierarchies are based on "is-a-part-of" relationships.

NB: It is possible to block the passing mechanism by prefixing the device name with a "@". In that case, only the specified device will execute the command.

Example:

```
group ab {a,b};
ab.n = 4;
a:a.n, b:b.n, ab:ab.n;
[09696528:a] 4.000000
[09696528:b] 4.000000
[09696528:ab] 4.000000
```
a.n = 1;
b.n = 2;
@ab.n = 5;
a:a.n, b:b.n, ab:ab.n;
[09696528:a] 1.000000
[09696528:b] 2.000000
[09696528:ab] 5.000000

For any robot, it is advised to design a hierarchy of devices, with *robot* on the top. This is usually done via a URBI.INI file (see 1.9).

### 1.8 Function definition

Functions can be defined in URBI 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. The parameters are always local to the function, like any non-global (i.e. without prefix) variable in the function body.

Example:

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

def print(x)
{
    echo x;
    return
}
```

Functions can return values or void. If they return values, they can be used in expressions. Recursive calls of functions are allowed in URBI, since all local variables and parameters are local to the function call (the server gives them a temporary prefix which is linked to the function call).

### 1.9 Files

If the robot has a filesystem available, it is possible to do some very simple file manipulation.

```urbi
load(file)
```

This command loads the file with name `file` given as a string, and push its content into the execution queue, as if the commands had been entered by hand.

```urbi
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.
The URBI.INI file, located in the root of the filesystem, is executed when the server starts. It is typically used to execute some initialization commands specific to the robot, like building the group hierarchy and playing a welcome sound.

The commands are executed in a virtual connection and any output is lost.

CLIENT.INI

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

1.10 Standard commands, operators or global variables

1.10.1 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.

The echo command can also be used to display system messages from one connection to the other. Simply use the syntax: echo <connectionTag> : <value>

Examples:

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

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

echo U597767000 : "hello"
/* equivalent to echo "hello" on
connection U597767000
*/
```

1.10.2 exec

exec(s) pushes the string s into the execution queue, as if it had been entered by hand.

Examples:

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

1.10.3 stop, block, unblock

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.
1.10.4 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 in the message header and can be used to do some accurate time measures on the server.

1.10.5 motoron, motoroff

The motoron command starts the motors (the joints or wheels are not loose anymore). The motoroff does the opposite.

1.10.6 killall, disconnect, quit

The commands killall and disconnect are followed by a connection identificator. Each client has a unique connection identificator, 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.

disconnect id will close the connection whose ID is id.

1.10.7 quit, reboot, shutdown

quit closes the current connection.

reboot reboots the robot and shutdown switches the robot off.

1.10.8 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.
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 current 1.0 version of URBI server for Aibo robot, as distributed on sourceforge (the number before the brackets is not part of the header, it is here just to simplify the line reference in the following):

```
1 [00139464:notag]
2 [00139464:start] ********************************************************************************
3 [00139464:start] URBI Language specif 1.0 - Copyright (C) 2004 JC Baillie
4 [00139464:start] URBI Kernel version 1.0
5 [00139464:start]
6 [00139464:start] URBI Server version 1.0 for Aibo ERS2xx/ERS7 Robots
7 [00139464:start] (C) 2004 Jean-Christophe Baillie
8 [00139464:start]
9 [00139464:start] URBI comes with ABSOLUTELY NO WARRANTY;
10 [00139464:start] This is free software, and you are welcome to redistribute
11 [00139464:start] it under certain conditions; see GNU GPL for details.
12 [00139464:start]
14 [00139464:start] ********************************************************************************
15 [00139464:ident] ID: U597766392
```

Line 1 is optional.
Line 2 and 14 are mandatory and all lines between them must have the start tag.
Line 3 is mandatory, specifying the corresponding version of the URBI Specification that the server implements and the copyright holder for the specification.
Line 4 is mandatory for URBI Servers who derive from the sourceforge GPL implementation, to make a clear reference to the kernel version used. The source code on sourceforge is divided into a kernel part and a robot-specific part. Each of these parts can have its own version number. Lines 6 and 7 can be used to specify the version of the robot-specific part and the type of robot supported.
Lines 5, 8 and 12 are optional. They are here just to ease the reading of the header.
Line 9 is mandatory.
Line 10 and 11 are mandatory for any server that has no specific license (which means that it is automatically a GPL license, as specified in the URBI Language License).
Line 13 is mandatory for any server with code that derives from the code on sourceforge.
Line 15 is mandatory, with a ident tag. It specify the tag of the connection (to be used for local variables prefix or killall, disconnect commands).

---

1The kernel contains all the code that is not specific to a robot (the majority of the code). The robot-specific part derives some robot-specific instances from the kernel object and instanciates some virtual members
Chapter 2

URBI grammar

2.1 Bison grammar

Here is the full URBI grammar which is part of the specification. Terminal symbols are written in upper case and are described in the following section.

```
ROOT:
  refvariable ASSIGN binary SEMICOLON
  | taggedcommands
  ;

taggedcommands:
  taggedcommand
  | taggedcommands COMMA taggedcommands
  | taggedcommands SEMICOLON taggedcommands
  | taggedcommands PIPE taggedcommands
  | taggedcommands AND taggedcommands
  ;

taggedcommand:
  command
  | IDENTIFIER flags COLON command
  ;

flags:
  /* empty */
  | flags FLAG
  ;

command:
  instruction
  | LBRACKET taggedcommands RBRACKET
  ;

instruction:
  /* empty */
  | NOOP
  | refvariable ASSIGN expr namedparameters
```
expr
| RETURN
| RETURN expr
| ECHO expr namedparameters
| ECHO IDENTIFIER COLON expr namedparameters
| GROUP IDENTIFIER LBRACKET identifiers RBRACKET
| GROUP IDENTIFIER
| ALIAS
| ALIAS variable variable
| OPERATOR
| OPERATOR_ID IDENTIFIER
| OPERATOR_VAR variable
| OPERATOR_ID_PARAM LSBRACKET MULTIPLEX RSBRACKET refvariable
| WAIT expr
| WAIT LPAREN softtest RPAREN
| refvariable MINUSMINUS
| refvariable PLUSPLUS
| DEF variable LPAREN identifiers RPAREN LBRACKET taggedcommands RBRACKET
| IF LPAREN test RPAREN taggedcommand
| IF LPAREN test RPAREN taggedcommand ELSE taggedcommand
| AT LPAREN softtest RPAREN taggedcommand
| AT LPAREN softtest RPAREN taggedcommand ONLEAVE taggedcommand
| AT AND LPAREN softtest RPAREN taggedcommand
| AT AND LPAREN softtest RPAREN taggedcommand ONLEAVE taggedcommand
| WHILE LPAREN test RPAREN taggedcommand
| WHILE AND LPAREN test RPAREN taggedcommand
| WHILE PIPE LPAREN test RPAREN taggedcommand
| WHILE PIPE LPAREN test RPAREN taggedcommand
| WHENEVER LPAREN softtest RPAREN taggedcommand
| WHENEVER LPAREN softtest RPAREN taggedcommand ELSE taggedcommand
| LOOP taggedcommand
| LOOPN LPAREN expr RPAREN taggedcommand
| LOOPN PIPE LPAREN expr RPAREN taggedcommand
| LOOPN AND LPAREN expr RPAREN taggedcommand
| FOR LPAREN instruction SEMICOLON test SEMICOLON
  | instruction RPAREN taggedcommand
| FOR PIPE LPAREN instruction SEMICOLON test SEMICOLON
  | instruction RPAREN taggedcommand
| FOR AND LPAREN instruction SEMICOLON test SEMICOLON
  | instruction RPAREN taggedcommand

array:
/* empty */
| LSBRACKET expr RSBRACKET array
|

purevariable:
| DOLLAR LPAREN expr RPAREN
| IDENTIFIER array
| STRUCT array
|
variable:
  purevariable
  | COLON purevariable
  ;

refvariable:
  DOLLAR LPAREN expr RPAREN
  | AROBASE DOLLAR LPAREN expr RPAREN
  | IDENTIFIER array
  | AROBASE STRUCT array
  | STRUCT array
  ;

namedparameters:
  /* empty */
  | IDENTIFIER COLON expr namedparameters
  ;

binary:
  BIN NUM
  | BIN NUM rawparameters
  ;

eexpr:
  NUM
  | STRING
  | TEST LPAREN test RPAREN
  | PERCENT variable
  | OPERATOR_ID_PARAM variable
  | FUNCTION_VAR IDENTIFIER
  | refvariable LPAREN parameterlist RPAREN
  | variable
  | expr PLUS expr
  | expr MINUS expr
  | expr MULT expr
  | expr DIV expr
  | COPY expr
  | MINUS expr
  | expr EXP expr
  | LPAREN expr RPAREN
  ;

parameterlist:
  /* empty */
  | parameters
  ;

parameters:
  expr
  | expr COMMA parameters
  ;
2.2 Terminal symbols and flex regexp

The following are the terminal symbols definitions, using regexp in the style of a flex definition file.

NUMBER \( (\.[0-9]+) | ([0-9]+(\.[0-9]*)?)) \)
ID \([a-zA-Z_][_A-Za-z0-9]*\)
STRING \"\([^{\"]|\\\\"\")*\"\)

Comment definition:

"//[^\n]*
"#[^\n]*
\/*/[^.^\n"]*\*/

Other terminal symbols:
"inf" : NUM (infinite value)
"pi" : NUM (value of pi)
"+report" : FLAG
"+error" : FLAG
"+end" : FLAG
"+begin" : FLAG

{NUMBER}"s" : TIMEVALUE
{NUMBER}"ms" : TIMEVALUE

{NUMBER} : NUM
{STRING} : STRING

stop |
unblock |
block |
killall |
disconnect : OPERATOR_ID

rangemin |
rangemax |
info |
unit : FUNCTION_VAR

delete : OPERATOR_VAR

mix |
normal |
cancel |
discard |
queue : MULTIPLEX

blend : OPERATOR_ID_PARAM

motoron |
motoroff |
ping |
devices |
vars |
uservars |
commands |
debugon |
debugoff |
quit |
reboot |
shutdown : OPERATOR

if : IF
else : ELSE
while : WHILE
at : AT
whenever : WHENEVER
onleave : ONLEAVE
for : FOR
loop : LOOP
loopn : LOOPN
group : GROUP
echo : ECHO
wait : WAIT
def : DEF
return : RETURN
"BIN" : BIN
bin : BIN
noop : NOOP
true : TRUECONST
false : FALSECONST
booleval : TEST
copy : COPY
alias : ALIAS

"&&" : ANDOPERATOR
"||" : OROPERATOR
";" : SEMICOLON
":" : COLON
"," : COMMA
"&" : AND
"-" : PIPE
"!" : BANG
"--" : MINUSMINUS
"++" : PLUSPLUS
"*" : MULT
"/" : DIV
"+" : PLUS
"^" : EXP
"-" : MINUS
"=" : ASSIGN
"==" : EQ
"~=" : REQ
"%==" : PEQ
"!=" : NE
">=" : GE
"<=" : LE
"~" : TILDE
"%" : PERCENT
"@" : AROBASE
"(" : LPAREN
")" : RPAREN
"[" : LSBRACKET
"]" : RSBRACKET
"{" : LBRACKET
"}" : RBRACKET
"$" : DOLLAR
{ID} : IDENTIFIER
(ID)}.{ID} : STRUCT
[ \t\n\r]+ /* eat up whitespace */