

GOSTAI

URBI QUICK START GUIDE

The logo for URBI, consisting of the letters 'URBI' in a bold, black, sans-serif font. The text is centered within a light blue rectangular box with a subtle gradient and a slight drop shadow.

Version 1.0
November 2007

Copyright © Gostai SAS
2006 - 2007

I.	<u>INTRODUCTION</u>	3
A.	PURPOSE	3
B.	URBI IN A FEW WORDS	3
C.	VOCABULARY	4
D.	URBI ARCHITECTURE SCHEMATICS	5
II.	<u>LAUNCHING AND CONNECTING TO AN URBI ENGINE</u>	6
A.	LAUNCHING AN URBI ENGINE	6
B.	CONNECTING TO AN URBI ENGINE	7
1.	INTERACTIVE CONNECTION	7
2.	REMOTE CONNECTION	7
3.	CONNECTION MESSAGE	7
III.	<u>THE URBI LANGUAGE</u>	9
A.	FOREWORDS	9
B.	KNOWN LANGUAGE FEATURES	10
C.	URBI LANGUAGE BUILTIN FEATURES	13
1.	TASKS AND PARALLELISM	13
2.	EVENTS	14
3.	TIME	15
4.	TAGS AND CODE EXECUTION HANDLING	16
5.	VARIABLE PROPERTIES	17
6.	OBJECT GROUPS AND BROADCASTING	18
7.	LOADING SCRIPTS	19
D.	SIMPLE BEHAVIOR SCRIPT EXAMPLE	19
IV.	<u>UOBJECT: THE PLUGIN ARCHITECTURE</u>	20
A.	FOREWORDS	20
B.	UOBJECT TEMPLATE	20
1.	TEMPLATE HEADER FILE	20
2.	TEMPLATE IMPLEMENTATION FILE	22
C.	BUILDING UOBJECTS	24
1.	BUILDING WITH MS VISUAL C++ ON WINDOWS	24
2.	BUILDING WITH DEV-C++ ON WINDOWS	24
3.	OTHER BUILDING METHODS	24
D.	MAKE A UOBJECT FROM AN EXISTING C++ CLASS	25
E.	DIFFERENCES BETWEEN REMOTE AND PLUGIN UOBJECTS	25
F.	EXTRA UOBJECT FEATURES	26
1.	UVARS AND UVALUES	26
2.	UTIMERS AND USETUPDATE	26
3.	NOTIFIES	27
V.	<u>MISSING PARTS AND INFORMATION</u>	28

I. Introduction

A. Purpose

This document is intended to provide every needed piece of information to start working with URBI, UObjects, liburbi and URBI utilities.

It is assumed that the reader has basic knowledge of writing computer programs in his environment. This can be under Microsoft Windows using MS Visual C++, MinGW or Dev-C++. It can also be under Linux and Mac OS X, using either GNU or proprietary tools.

The reader is also required to know a little about C/C++ programming. Being familiar with scripting languages as Python or Ruby may also be of help.

Only basic programming skills are needed for reading this document. But it is not intended to total beginners.

If the reader needs more advanced or specific documentation, it is available on Gostai's website at the following address: www.gostai.com/doc.

This document is still at a work-in-progress stage. Please report any error, or missing information at support@gostai.com.

Sections specific to operating systems, examples, and more advanced features will be added soon.

B. URBI in a few words

URBI is a new interpreted (script) language. It was developed first for the Aibo (made by Sony), and was later extended in order to become a universal development platform for any robotic system.

It is also possible to use URBI for non-robotic applications, such as video games, intelligent houses, alarm systems and many more.

These are the core language builtin features making URBI the right choice for developing these kinds of applications:

- Parallelism
- Task and event handling
- Time handling
- Code tagging
- UObject architecture (plugins)
- Client/Server architecture

These features come in other languages, libraries or development tools, but never all at the same time, neither with being so easy to use.

URBI aims at being used as a universal development platform for robotics. Hence, simplicity was not chosen over expressivity or powerfulness.

C. Vocabulary

As any interpreted programming language, URBI is not only a language, but also the corresponding virtual machine used to run URBI scripts. You can compare it to the Java virtual machine, or to a Python interpreter.

URBI goes further than many interpreted languages in the way that it lets you build your own virtual machine, enhance it with plugins and make other applications converse with it.

This brings the need to introduce some vocabulary:

- **URBI Kernel:** the URBI kernel is the heart of URBI. It is the library holding all the core features needed to execute user scripts and requests.
- **URBI Cores:** an URBI core is a kernel linked to system specific functions needed to run URBI.
- **URBI Engines:** An URBI engine is what may also be called an URBI server. It is a full program, on which scripts can be executed. This server can be used directly on a robot or a computer, and receive connections from clients. An URBI Engine is an URBI core linked with any number of UObjects. This is what most users will encounter first.
- **liburbi:** the liburbi is a C++ (for now) library, used to allow any program to send commands to an URBI server, and to receive its answers. It brings URBI to your applications.
- **UObjects:** a UObject is a plugin. It is mostly a C++ (for now) class, usable by URBI, either as a component of a server (a plugin UObject), or as a stand alone application, connected to an URBI server (a remote UObject).
- **Connection:** as said earlier, URBI is a network application. A server (if configured to do so) can accept incoming connections from any computer. A connection will design a channel through which a client (an application or an interactive user shell) send commands to URBI and receives answers. An URBI server can handle an arbitrary number of network connections from local or remote clients.

D. URBI architecture schematics

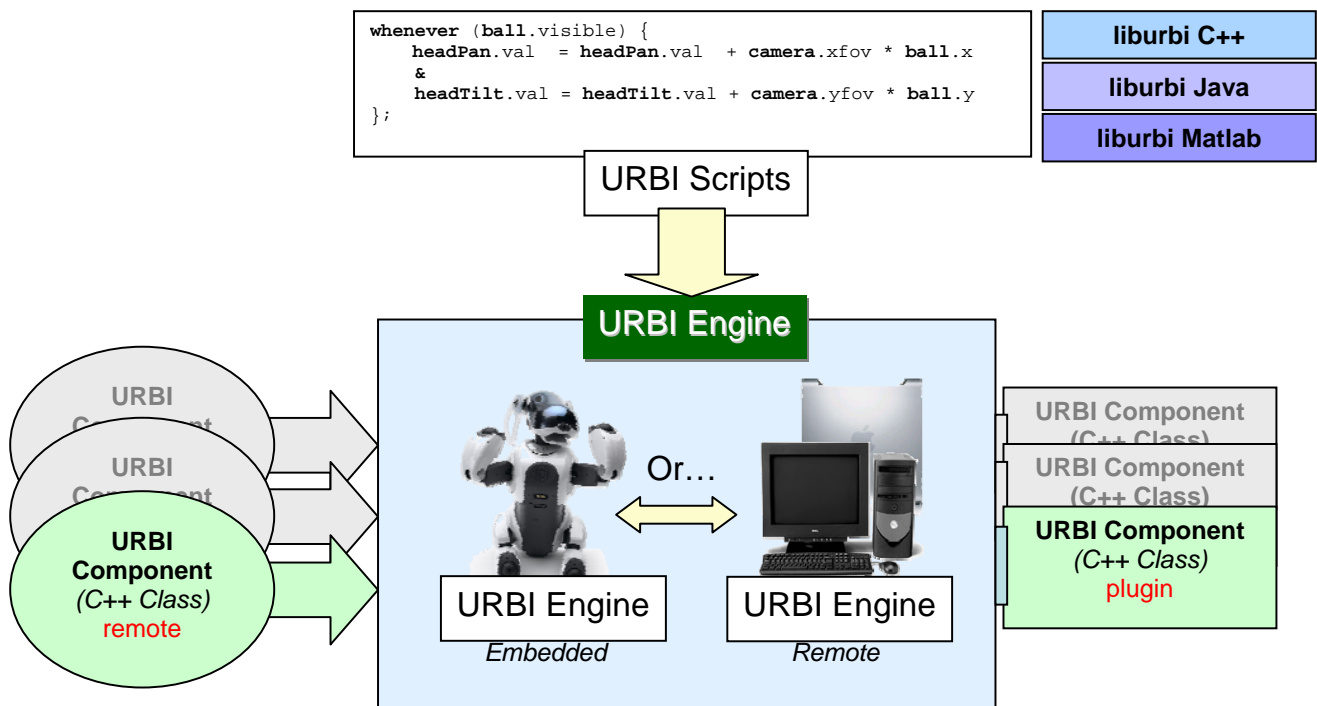
The figure below represents the way URBI components interact with each other.

The URBI Engine is the heart of the system. It can run on a robot (embedded) or on a computer (remote). In case of a remote engine, the engine is responsible for forwarding requests and answers to and from the robot. In other system, a robot may not be used (alarm system) or the engine can be part of another application (video game).

The standard URBI Engine can be augmented with plugins. All these plugins are C++ classes (for now), and can be either standalone applications connected to the engine through a network (remote UObjects), or integrated into the engine when it is generated (plugin UObjects).

An URBI Engine can transparently use any feature or functionality provided by a UObject as if it was part of the same application from the beginning.

Once the URBI Engine is running, it is possible to connect to it, and to send commands. This is done with a user shell (directly into the server builtin shell or through telnet connections), or any with application using the liburbi to send and receive messages.



II. Launching and connecting to an URBI Engine

A. Launching an URBI Engine

This is the help message you get when executing an URBI Engine without any command line options.

```
usage: urbi -server [OPTIONS] PERIOD [PATHS... ]

PERIOD  base URBI interval in milliseconds
PATHS   absolute or relative path elements searched
        in order for files when 'load' is called.

Options:
-p PORT  specify the tcp port URBI will listen to.
-b ADDR  bind to a specific ip address.
-w FILE  write port number to specified file.
-n       disable networking.
-r       report the time taken by URBI loop to execute
-i       treat stdin as an urbi connection
-s PERIOD shell-mode (no network) with given period
-t       use high priority scheduling
-f       fast mode: the server will run as fast
        as possible and emulate the period specified
-v       print version information and exit
--versi on
```

The important options to remember are:

- **-p**: used to change the port on which URBI listens. 54000 by default.
- **-b**: URBI does not allow remote connections by default. Use `-b 0.0.0.0` to allow connections from any computer, or tune the option argument to match you need.
- **-i**: enable input reading from terminal. This allows to type URBI commands right into the engine without remote connection.
- **PERIOD**: is the time a cycle will last. Basically, URBI will execute commands every PERIOD milliseconds.
- **PATHS**: is a list of directory name in which URBI will try to load requested files in the future. Defaults to current directory.

A standard way to call an URBI Engine for testing purpose is:

```
urbi -server -i -b 0.0.0.0 10
```

This will launch an URBI server allowing remote connections, interactive shell and running commands every 10 milliseconds.

B. Connecting to an URBI Engine

1. Interactive connection

In case the server which you want to connect to was launched with the `-i` options, you can type commands directly into the command window or shell where the server is running.

In this particular case, it can be useful to launch the server with **rlwrap** (readline wrapper) or any equivalent tool to gain proper shell navigation and history. These features are not yet available in the standard URBI interactive connection.

2. Remote connection

A “remote” connection can be run either on the same machine as where the server is executed, or from any other computer (provided this computer can access to the other, and that the server is running with the right `-i` option).

The easiest way to connection to an URBI server is to use a telnet client. This should be available under every operating system.

For the same reasons as for the interactive connection, it can be wise to use tools such as **rlwrap**.

Some graphical and more user-friendly clients can be found on www.sourceforge.net.

Note that Gostai will soon release its URBI Studio (<http://www.gostai.com/studio.html>). This development suite will provide a remote usable with every URBI server, a behavior editor, and an animation editor.

3. Connection message

Upon connection, a client receives the following message:

```
*** *****
*** URBI Language speci f 1.0 - Copyright 2005-2007 Gostai SAS
*** URBI Kernel versi on 1.0 rev. 2121
***
***   URBI Engi ne versi on 1.0 rev. 042
***   (C) 2006-2007 Gostai SAS
***
*** URBI comes wi th ABSOLUTELY NO WARRANTY;
*** Thi s software can be used under certain condi tions;
*** see LICENSE fi le for detai ls.
***
*** See http: //www. urbi forge. com for news and updates.
*** *****
Ready.
```

This message confirms the connection to the server is well established. It indicates the URBI version used by this engine (1.0), and the build revisions for both URBI Kernel library and URBI Engine.

Two more messages can be printed:

```
URBI: Invalid or missing license.  
URBI engine will stop after five minutes.
```

And

```
*** ID: U5267456
```

The first message is a warning displayed when your URBI license is wrong or has expired. The license file is to be named **urbi.key** and placed in the same directory as the engine binary. Without a proper license file, your server will stop after five minutes of execution.

The second message is displayed when using a remote connection. It indicates the connection ID given to your connection. We will see later how to use it.

III. The URBI language

A. Forewords

URBI was designed in order to look familiar to programmers used to C/C++ programming. For this reason, programmers also used to Python, Ruby, JavaScript, Pascal or other imperative object oriented language should be quickly able to write URBI scripts.

As URBI is an interpreted language, it is not necessary to compile your scripts. They can be directly type into a connection. Your programs can also be modified at runtime (while executing) with later commands.

In the remainder of this chapter, we will write URBI short commands to describe how to use the core language. It is considered that a running URBI server is available and freshly started (so no older commands will interfere with the following examples).

The following is the simplest command you can type into you connection and its associated answer:

```
0;  
[00009587: notag] 0.000000
```

This command sends the value **0** to the server. The server answers “**0**”.

The answer should be read “at time 00009587, I received a command which result was the value 0.000000. This command was not tagged.”

The time is expressed in millisecond, 0 being the launch time of the server.

The “**notag**” tag is the default value when no tag is specified for a command (this default tag will not be printed in future versions of URBI).

B. Known language features

As said earlier, URBI is a language close to C++.

The following code should be self explanatory. If it not, please refer to the URBI tutorial found at the address given earlier.

```
0;
[00013169: notag] 0.000000
var n = 0;
n;
[00026664: notag] 0.000000

if (n > 0)
  echo "n > 0"
else
  echo "n <= 0";
[00099107: notag] *** n <= 0

i = 0;
while (i <= 2)
{
  echo i;
  i++;
};
[00149916: notag] *** 0.000000
[00149936: notag] *** 1.000000
[00149946: notag] *** 2.000000

for (i = 0; i <= 2; i++)
  echo i;
[00170484: notag] *** 0.000000
[00170504: notag] *** 1.000000
[00170514: notag] *** 2.000000
```

The previous script shows basic constructions. They should look familiar. URBI also introduces “**loop**” and “**loopn**”. These are an infinite loop, and a loop iterating “**n**” times.

For more complex instructions, functions can be defined and strings used:

```
function f (n)
{
  echo "value is : " + n;
};

f(1);
[00635142: notag] *** value is : 1
```

It is also possible to work with lists and arrays:

```
foreach i in l
{
  echo i;
};
[00798218: notag] *** 0.000000
[00798218: notag] *** 10.000000

a[0] = "value 0";
a[1] = 1;
a[1];
[00869513: notag] 1.000000
```

Please note that “**l**” is a list and “**a**” an array. In URBI 1.0, arrays don’t have a real existence. “**a[0]**” is a link to “**a__0**”. Thus, the first element of “**l**” cannot be accessed with “**l[0]**” (which is a link to “**l__0**”).

URBI also allows declaring classes and objects:

```
class C
{
  var val;
  function f ();
};

function C.f ()
{
  echo "function C.f";
};

C.f ();
[01214922: notag] *** function C.f

D = new C;
D.f ();
[01230463: notag] *** function C.f

function D.f ()
{
  echo "function D.f";
};

D.f ();
[01256403: notag] *** function D.f
```

This shows how to declare a class “**C**”, to implement its methods and to create objects from this class. It also demonstrates how to extend this class or to reimplement methods inherited from mother classes. Please note that URBI is not a real object oriented language, but a “**prototype-based language**”. If you are not familiar with this notion, it is not really important for beginning. You may think of it as putting “**static**” everywhere in you C++ code and confounding class and object.

URBI being interpreted, nothing is frozen in your code. You can add methods and attributes to classes or objects at anytime, and you can remove some others. You can create new classes or objects, add inheritance...

By default, any method or attribute inherited from a mother class is a link to it, not a copy. The previous example demonstrated it for methods, here is an example for attributes (same classes as above):

```
C.val = 1;
D.val ;
[01230463: notag] 1.000000
C.val = 2;
D.val ;
[01231254: notag] 2.000000
D.val = 3;
C.val ;
[01232395: notag] 2.000000
D.val ;
c
```

By default, any identifier you define is local to you connection (if in root level), or to the function being declared (same as a C++ file).

Any object is declared global, hence usable by anyone connected to the server.

If you want to make a variable or a function “**global**”, you have to prefix it like this:

```
global.myvar = "a global string";
```

In URBI, “**global**” means accessible for any user connected from anywhere.

C. URBI language builtin features

In this section, you may use the “**stopall**” command to stop any running URBI command.

1. Tasks and parallelism

Controlling a robot often means ordering it to perform tasks. In most programming languages, it is need to implement a task handler or scheduler to be able to ask the system for multiple task execution. Threads are often needed to perform multiple actions simultaneously.

These are hard to implement, and are not what you want to implement yourself. This is why URBI comes with builtin parallelism.

A basic task is an URBI command, typically a method call or a loop.

While C++ only provides “;” to separate commands, URBI comes with three more command separators. The comma (“,”) is a separator use to put a command in background. More precisely it allows launch other commands before the previous one finishes. These new command will be executed in parallel with the first one, and not queued as in other languages. Here is an example:

```
loopn (2) echo "A"; loopn (2) echo "B";

[01042640: notag] *** A
[01042660: notag] *** A
[01042670: notag] *** B
[01042680: notag] *** B

loopn (2) echo "A", loopn (2) echo "B";

[01052466: notag] *** A
[01052476: notag] *** B
[01052486: notag] *** A
[01052486: notag] *** B
```

The first pair of loops is executed sequentially, printing “A A B B”. The second one simultaneously executes its two loops, printing alternatively an “A” or a “B”.

URBI also introduces the operator “|” and the operator “&”. The “|” is the same as the “;”, except that the meaning of “A | B” is “as soon as A finishes, start B”. The “&” in “A & B” is the same as the “;” but meaning “start A and B at exactly the same time”.

“|” and “&” and *stronger* versions of “;” and “,”.

Please note that the “,” is useful when using a loop instruction. Without it, your connection will be stuck in the loop and you won’t be able to send more commands.

2. Events

In the same idea of controlling a robot, it appeared that a robot must be able to react to events, and to repeat tasks according to time values.

As above, implementing system answering these problems is a hard and long work.

URBI comes with intuitive features to declare events, and then to handle them.

The two base constructs used for event handling are “**at**” and “**whenever**”.

```
n = 0;
at (n > 0) echo "n i s p o s i t i v e";
n = 1;
[01889465: notag] *** n i s p o s i t i v e
```

```
n = 0;
whenever (n > 0) echo "n i s p o s i t i v e";
n = 1;
[01889465: notag] *** n i s p o s i t i v e
[01889475: notag] *** n i s p o s i t i v e
[01889485: notag] *** n i s p o s i t i v e
. . .
```

The “**at**” construct triggers once when the condition becomes true. The “**whenever**” at each cycle the condition is true. They respectively accept “**onleave**” and “**else**” constructs for when the condition respectively becomes or is false.

Events can also be declared and emitted:

```
event ev;
at (ev) echo "ev detected";
emit ev;
[02159243: notag] *** ev detected
```

3. Time

Same as for events, URBI comes with constructs to schedule tasks, and to handle time.

```
every (1s) echo time ();  
[00041254: notag] *** 41257.000000  
[00042257: notag] *** 42257.000000  
[00043265: notag] *** 43265.000000  
[00044272: notag] *** 44272.000000  
. . .
```

The previous example shows how to schedule the printing of the server running time every one second.

When not given any units, time values are in milliseconds. The can takes forms as “20s”, “3m” or “1d2h3m”.

Time features can also be use to test event durations, or to make assignments “last” some time:

```
at (n > 0 ~ 2s) echo "n > 0 for 2 seconds";  
n = 0;  
n = 2 & time ();  
[00624204: notag] 624205.000000  
[00626220: notag] *** n > 0 for 2 seconds
```

```
n = 0;  
every (100) echo n & n = 10 time:0.5s,  
[00723762: notag] *** 0.000000  
[00723871: notag] *** 2.040000  
[00723973: notag] *** 4.220000  
[00724073: notag] *** 6.220000  
[00724174: notag] *** 8.240000  
[00724279: notag] *** 10.000000  
. . .
```

The “~” operator indicates needed duration for a condition validation, while the “**time**” modifier indicated the duration of an assignment. Please refer to URBI specification or tutorial for full list of modifiers.

Other time constructs are “**wait**” and “**waituntil**”:

```
time () | wait (1s) | echo "1s later";  
[00993986: notag] 993987.000000  
[00994990: notag] *** 1s later  
  
waituntil (n == 5) | echo "n == 5",  
  
n = 5;  
[01024855: notag] *** n == 5
```

4. Tags and code execution handling

Up to now, we have been able to declare tasks, events, durations, times and launch commands. We are also able to schedule commands.

But this is not enough to write complex behaviors for a system (robotic or not).

To fulfill this goal, URBI come with code designation and manipulation constructs and code labels called “**tags**”.

To tag a command, simply prefix it with a “**mytag:**”:

```
mytag: echo "tagged command";  
[00011824: mytag] *** tagged command
```

Note that the header of the server answer no longer displays “**notag**” but “**mytag**” which is the specified tag in the command.

This allows to *name* any command. Once, a command is named, it is possible to interact with it:

```
myloop: every (1s) echo "loop",  
wait (2s) | stop myloop,  
  
[00195488: notag] *** loop  
[00196490: notag] *** loop
```

The script above launches an infinite loop naming it “**myloop**”. Just after, a command is launched saying “stop the code name **myloop** in 2 seconds”. We can see the loop perform to iterations then stop. Note that the printing commands are not tagged while the loop is (“**notag**” is printed).

There are times when you don’t want to *stop* code as it can be useful sometime in the future. In this case, you can use the “**freeze**” and “**unfreeze**” keywords. While when stopped, your code is definitively lost, when frozen, it can be resumed at any time.

You can also use other keywords as “**stopif**”, “**freezeif**” or “**timeout**” to control you code. These keywords are used to prefix commands in this way:

```
timeout(3s) myloop: every (1s) echo "loop";  
[00620685] *** loop  
[00621686] *** loop  
[00622689] *** loop
```

5. Variable properties

In robots, variables may not be just simple values. It may be useful to specify properties of these values.

This is done like in the following script:

```
n = 0;  
n->rangemax = 100;  
n = 50;  
n;  
[00811915: notag] 50.000000  
n = 200;  
n;  
[00817458: notag] 100.000000  
  
n->blend = "mix";  
n = 50 & n = 20;  
n;  
[00849544: notag] 35.000000
```

This example show how to set limits to a value (“**rangemin**” also exists). This can be useful for a motor speed.

The blendmode (“->**blend**”) is an URBI feature provided to handle simultaneous assignments to variables. In this case, the values 50 and 20 are put into the variable “**n**” at the same time, resulting in a value of 35 (being their average).

Please refer to URBI specification for full blendmode list.

6. Object groups and broadcasting

A robot is often a multipart system. It also often has parts of the same type, or with which you may want to interact in the same way.

For this, URBI introduces object grouping and call broadcasting.

```
class C { var val ; };
c1 = new C; c1.val = 1;
c2 = new C; c2.val = 2;
c3 = new C; c3.val = 3;

group g {c1, c2, c3};

function C.f () { echo val ; };

g.f ();

[00015429: notag] *** 3.000000
[00015429: notag] *** 2.000000
[00015429: notag] *** 1.000000
```

The example above declares a class and three objects from this class, each having a different value for its “**val**” attribute.

It then declares a group “**g**”, adds a method to the mother class of our objects, and calls this method through the group. The result is a call on each of the object of the group.

You can imagine the result of a command like “**g.val = 4;**”.

The groups can also be used with “**foreach**” loops:

```
foreach o in group g
{
  echo $o.val ;
};

[00218050: notag] *** 1.000000
[00218050: notag] *** 2.000000
[00218050: notag] *** 3.000000
```

Note the “**\$**” used to access the foreach iterator.

7. Loading scripts

It is possible to store your scripts in text file to reload them later. For this, use the “**load**” command in an URBI connection.

Remember that files are searched in directories given on the server command line.

```
load ("my_file.u");
```

The URBI server will warn you if the file is not found or contains errors.

Note that the script is loaded in you current connection. Using a “;” after a load instruction may help preventing losing control after loading an infinite script.

D. Simple behavior script example

The following script is made to run on a Sony Aibo:

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

freeze ball tracking;

at (headSensor.val > 0 ~ 2s)
{
  echo "ball tracking running";
  unfreeze ball tracking;
};

at (backSensorM.val > 10 ~ 1s)
{
  echo "ball tracking frozen";
  freeze ball tracking;
};
```

This simple script uses most of URBI features: events, tags, durations, code control...

This script also uses Aibo builtin objects. These objects (plugin UObjects in this case), are C++ classes, linked with the URBI kernel, and made usable in URBI. They represent the hardware devices of the robot (motors, sensors) or software agents (ball).

These UObjects should be included in the URBI Engine by the robot manufacturer or by the engine developer if not the same. User can then add their own UObjects, or enhance existing ones.

IV. UObject: the plugin architecture

A. Forewords

As said before, a UObject is an URBI plugin. It can either be linked with the URBI Kernel to be directly usable in the resulting engine, or compiles as a standalone application. In the later case, the resulting “**remote UObject**” need to connect to an URBI server to be usable. This allows to process complex tasks (as face recognition) on a powerful computer while the simpler behaviors run directly on the robot.

B. UObject template

The following is a minimalist example of UObject. It is composed of a header file, and of the object definition. This template is enough to be make a usable UObject in any URBI Engine. We will see later how to add functionalities.

1. Template header file

a) Source

```
#ifndef UOB_TEMPLATE_HH_
# define UOB_TEMPLATE_HH_

# include "urbi/uobject.hh"

class uob_template : public urbi::UObject
{
public:
    uob_template (const std::string& __name);
    virtual ~uob_template ();

    int init ();

    int f ();

    urbi::UVar val;
};

#endif /* !UOB_TEMPLATE_HH_ */
```

b) Line by line explanation

Include UObject definition provided by Gostai:

```
# include "urbi/uobject.hh"
```

A UObject must derive from urbi::UObject:

```
class uob_template : public urbi::UObject
```

Any method or attribute you want to access from URBI must be public:

```
public:
```

This is the default UObject constructor. The “__name” string is the URBI name of the object. A call like “uob = new uob_template;” will call this C++ constructor with “uob” as argument:

```
uob_template (const std::string& __name);
```

This virtual destructor is not required, but may be useful sometimes:

```
virtual ~uob_template ();
```

The “init” method is automatically called by URBI after a call to “new”. The C++ constructor is called to create the object in memory. The URBI constructor is then called to init this object. This init method can be (re)defined in URBI, while the C++ cannot:

```
int init ();
```

urbi::UVar is the default type for object attributes. This special type can hold any value, from integers to jpeg images:

```
urbi::UVar val;
```

You can in theory use any C++ code in a UObject but some specific limitations exist. The major ones will be detailed in the following sections.

Feel free to declare any protected method or attribute which may be needed by your “published” URBI interface for a UObject.

2. Template implementation file

a) Source

```
#include "uob_template.hh"

UStart (uob_template);

uob_template::uob_template (const std::string& __name)
    : UObject (__name)
{
    UBindFunction (uob_template, init);
}

uob_template::~uob_template ()
{ }

int
uob_template::init ()
{
    UBindFunction (uob_template, f);
    UBindVar (uob_template, val);

    val = 0;

    return 0;
}

int
uob_template::f ()
{
    static int cpt = 0;

    return cpt++;
}
```

b) Line by line explanation

Include you UObject header:

```
#include "uob_template.hh"
```

This UStart command tells URBI that this C++ class is to be used in URBI. Your UObject won't be visible if omitted.

```
UStart (uob_template);
```

This is the C++ constructor. As said before it takes the URBI object name as argument. This name is forwarded to the UObject class.

```
uob_template::uob_template (const std::string& __name)
: UObject (__name)
{
```

In this C++ constructor, we tell URBI to use this class init method. The method is “**binded**” to its corresponding URBI name (uob_template.init) and will be called when new uob_template UObjects will be created.

```
UBindFunction (uob_template, init);
}
```

This is the init method which we talked about. It is used to bind UObject methods and attributes, and if needed to initialize them.

This method must return 0 on success. Any other value will be considered an error, and the object will be automatically deleted:

```
int
uob_template::init ()
{
    UBindFunction (uob_template, f);
    UBindVar (uob_template, val);

    val = 0;

    return 0;
}
```

This is a simple C++ method, binded in “**init**” hence usable in URBI scripts:

```
int
uob_template::f ()
{
    static int cpt = 0;

    return cpt++;
}
```

C. Building UObjects

Once the code of your UObject is written, you still need to make a program from it.

1. Building with MS Visual C++ on Windows

To build a UObject in MS Visual C++, the only needed steps should be to add the corresponding library in the linker options, and the right folder in the include directory list.

The library used by your linker should be:

liburbicore.lib/.a: if you want to build an autonomous URBI Engine

libkernel-remote.lib/.a: if you are building a remote UObject

These libraries are located in the “gostai/core” directory or your URBI distribution. The “uobject.hh” file should be in the “include” directory of your URBI distribution.

To build an URBI Engine, you will also need to link your objects with **Iphlpapi.dll** and **libeay32MT.dll** found at:

<http://www.slproweb.com/products/Win32OpenSSL.html>

Be sure than the following option is set in visual studio configuration:

Configuration project - C/C++ - Code generation - RuntimeLibrary - Multithread dll (/MD)

Gostai distributes Visual C++ template projects to help with these first steps. They should also be within you URBI distribution.

Warning: MS Visual C++ prevents from binding methods returning void. The simple workaround is to make your binded methods return a dummy value (as an integer).

2. Building with Dev-C++ on Windows

The steps to follow to compile with Dev-C++ are the same as with MS Visual C++.

Project templates are also available.

3. Other building methods

This last method is available for Linux, Mac OS X and MS Windows (using cygwin or MinGW).

Your URBI distribution should hold a few tools named umake and umake-something. This is a compact build system using freely available tools to build UObjects.

Simply call “**umake-engine**” to build a full URBI engine from source files. Or call “**umake-remote**” to build a remote UObject from the very same files.

These two commands are wrappers around “**umake**”. Have a look at their sources if you want.

Use “**umake -help**” or the umake manual for full help.

The important options are:

- -o, --output: to specify the output name
- -c, --clean: to force full recompilation
- --core, --host: more advanced features handled by the wrappers.
- EXTRA_CPPFLAGS="", EXTRA_LDFLAGS="": flags to tell umake which flags and libraries are needed to build your UObjects.

D. Make a UObject from an existing C++ class

To transform an existing class into a UObject, the simplest method is to create a new class, inheriting from both UObject and your class.

Add the UStart, the call to UObject constructor and the binding on “**init**” in your constructor.

Add the init method, and bind any method or attribute from your class you need.

The following limitations apply for now:

- URBI don't support overloading, methods must have unique names and prototypes
- URBI cannot bind “const” methods
- It is not possible to bind methods returning void in MS Visual C++

E. Differences between remote and plugin UObjects

When using remote or plugin UObjects, you have to remember where the code is executed. A plugin UObject is part of the engine, while a remote UObjects is another distinct binary.

The one thing to know about plugin UObjects is that the URBI Kernel has a single thread for command execution. It means than a C++ method call (to a binded method) will freeze the rest of the kernel until it has returned. For this reason, method calls during a “long time” can be dangerous for the engine time and event handlers. They will not be running as long as a method call is running.

An URBI method call (one written in pure URBI) does not have this problem as interpreted code can be stopped and resumed from anywhere at anytime. But C++ is faster.

Calls to methods binded in remote UObjects will not cause this problem. As the remote UObject is running in another process, a call to one of its method will only freeze this process and not the whole URBI Engine.

Some latency can be introduced by remote UObjects as data is send by TCP/IP over the network.

F. Extra UObject features

1. UVars and UValues

You may have noticed the special type `urbi::UVar` was used in the `UObject` template above. This type is used as a variable container in URBI.

You can add any number of UVars to you UObjects and bind them. Assign operators have been overloaded so you can put any valid URBI value into an UVar. So are the cast operators to get the values back.

The `urbi::UValue` type holds the variable values used by URBI. They have the same operator as the UVars.

When you bind a method in a UObject, you can declare its arguments either as UValues, or as their real type (string, int...). Do not bind methods with UVars as arguments, this may have unexpected behavior.

Have a look at UObject documentation for more information. Just keep in mind that any value in C++ is usable in URBI and any URBI value usable in C++.

2. UTimers and USetUpdate

It is possible to schedule UObject methods call as it is done with the “**every**” URBI keyword. You can use “**USetUpdate**” which will call the object “**update**” method with the time period given as its argument. You can also use “**USetTime**”, calling the given methods every given period.

In `uob_template::init`, add:

```
USetUpdate (100);
USetTimer (500, &uob_template::f);
```

And a method update such as:

```
int
uob_template::update ()
{
    return 0;
}
```

in the definition file (and its prototype in the header file).

3. Notifies

Notifies are callback functions which can be set on Object's UVars.

You can put notifies on read access or on write access on UVars (or both).

Add this to your files:

```
/* In .hh file */
int on_val change (urbi::UVar& val);
```

```
/* In .cc file */
// In uob_template::init
UNotifyChange (val, &uob_template::on_val change);

// New method
int
uob_template::on_val change (urbi::UVar& val)
{
    std::cerr << "uob_template.val is now : "
                << (int)val << std::endl;
    return (int)val;
}
```

Please refer to UObject documentation for complete information about notifies.

V. Missing parts and information

This document still lacks information about the liburbi and how to integrate it within an existing application.

It is also needed to add more UObject and URBI scripts example, with use cases and build examples.

Some URBI features may also be explained with more accurate and extensive details extracted from the respective URBI documentations.

If you have in mind something you may want in this document, feel free to tell us a support@gostai.com.

This document will be updated soon.