



The Urbi Software Development Kit

Version 2.7.5

Gostai

January 30, 2012
(Revision 2.7.5)

Urbi is an open source robotics software platform. It is cross-platform: it supports several robots (Gostai Jazz, Segway RMP, Pioneer 3-DX, LEGO Mindstorms NXT, Aldebaran Nao...) and simulators (Webots...). It runs on top of all the major operating systems. It features a C++/Java middleware, UObject, to interface components such as motors, cameras, and algorithms; and an innovative scripting language, urbiscript, with built-in support for parallel and event-based programming. It is compatible with ROS, the Robotic Operating System.



Chapter 1

Introduction

Urbi SDK is a fully-featured environment to orchestrate complex organizations of components. It is an open source framework for robotics and other complex systems. It relies on a middleware architecture that coordinates components named UObjects. It also features urbiscript, a scripting language tailored to write orchestration programs.

1.1 Urbi and UObjects

Urbi makes the orchestration of independent and concurrent components easier. It was first designed for robotics: it provides all the needed features to coordinate the execution of various components (actuators, sensors, software devices that provide features such as text-to-speech, face recognition and so forth). Languages such as C++ are well suited to program the local, low-level, handling of these hardware or software devices; indeed one needs efficiency, small memory footprint, and access to low-level hardware details. Yet, when it comes to component orchestration and coordination, in a word, when it comes to addressing concurrency, it can be tedious to use such languages.

Middleware infrastructures make possible to use remote components as if they were local, to allow concurrent execution, to make synchronous or asynchronous requests and so forth. The *UObject* C++ architecture provides exactly this: a common API that allows conforming components to be used seamlessly in highly concurrent settings. Components need not be designed with UObjects in mind, rather, UObjects are typically “shells” around “regular” components.

Components with an UObject interface are naturally supported by the urbiscript programming language. This provides a tremendous help: one can interact with these components (making queries, changing them, observing their state, monitoring various kinds of events and so forth), which provides a huge speed-up during development.

Although made with robots in mind, the UObject architecture is well suited to tame any heavily concurrent environment, such as video games or complex systems in general.

1.2 The Big Picture

The [Figure 1.1](#) shows the architecture of Urbi. Let’s browse it bottom up.

At the lowest level, Urbi requires a (possibly very limited) embedded computer. This is the case for most robots today, but on occasion, some device cannot even run reasonably small pieces of code. In that case, Urbi can still be used, but then the robot is actually remote-controlled from a computer running Urbi.

Right on top of the hardware, is running the *Operating System*. Urbi supports the major OSes; it was also ported on top of real-time OSes such as Xenomai, and on specific OSes such as Aperios, Sony’s proprietary system running its Aibo robotic dog.

The *Urbi Runtime*, which is the actual core of the system, also known as the *engine* or the *kernel*, is interfacing the OS with the rest of the Urbi world, urbiscript and UObjects.

UObjects are used to bind hardware or software components, such as actuators and sensors on the one hand, and voice synthesis or face recognition on the other hand. They can be run locally on the robot, or on a remote, more powerful, computer.

To orchestrate all the components, urbiscript is a programming language of choice (see below).

Finally, applications are available for the Urbi environment. For instance, Gostai Studio provides high-level tools to develop complex robotic behaviors.

1.3 Urbi and urbiscript

urbiscript is a programming language primarily designed for robotics. It's a dynamic, prototype-based, object-oriented scripting language. It supports and emphasizes parallel and event-based programming, which are very popular paradigms in robotics, by providing core primitives and language constructs.

Its main features are:

- syntactically close to C++.

If you know C, C++, Java, or JavaScript, you can easily write urbiscript programs.

- fully integrated with C++.

You can bind C++ classes in urbiscript seamlessly. urbiscript is also integrated with many other languages such as Java, MatLab or Python.

- object-oriented.

It supports encapsulation, inheritance and inclusion polymorphism. Dynamic dispatching is available through monomethods — just as C++, C# or Java.

- concurrent.

It provides you with natural constructs to run and control high numbers of interacting concurrent tasks.

- event-based.

Triggering events and reacting to them is absolutely straightforward.

- functional programming.

Inspired by languages such as LISP or Caml, urbiscript features first class functions and pattern matching.

- client/server.

The interpreter accepts multiple connections from different sources (human users, robots, other servers ...) and enables them to interact.

- distributed.

You can run objects in different processes, potentially remote computers across the network.

1.4 Genesis

Urbi was first designed and implemented by Jean-Christophe Baillie, together with Matthieu Nottale. Because its users wildly acclaimed it, Jean-Christophe founded Gostai, a France-based Company that develops software for robotics with a strong emphasis on personal robotics.

Authors Urbi SDK 1 was further developed by Akim Demaille, Guillaume Deslandes, Quentin Hocquet, and Benoît Sigoure.

The Urbi SDK 2 project was started and developed by Akim Demaille, Quentin Hocquet, Matthieu Nottale, and Benoît Sigoure. Samuel Tardieu provided an immense help during the year 2008, in particular for the concurrency and event support.

The maintenance is currently carried out by Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Jean-Christophe Baillie is still deeply involved in the development of urbiscript, he regularly submits ideas, and occasionally even code!

Contributors Many people contributed significantly to Urbi, including Alexandre Morgand, Romain Bezut, Thomas Moulard, Clément Moussu, Nicolas Pierron.

1.5 Outline

This multi-part document provides a complete guide to Urbi. See [Chapter 35](#) for the various notations that are used in the document.

Part I — Urbi and UObjects User Manual

This part covers the Urbi architecture: its core components (client/server architecture), how its middleware works, how to include extensions as UObjects (C++ components) and so forth.

No knowledge of the urbiscript language is needed. As a matter of fact, Urbi can be used as a standalone middleware architecture to orchestrate the execution of existing components.

Yet urbiscript is a feature that “comes for free”: it is easy using it to experiment, prototype, and even program fully-featured applications that orchestrate native components. The interested reader should read either the urbiscript user manual ([Part II](#)), or the reference manual ([Chapter 23](#)).

Chapter 4 — Quick Start

This chapter, self-contained, shows the potential of Urbi used as a middleware.

Chapter 5 — The UObject API

This section shows the various steps of writing an Urbi C++ component using the UObject API.

Chapter 6 — The UObject Java API

UObjects can also be written in Java. This section demonstrates it all.

Chapter 7 — Use Cases

Interfacing a servomotor device as an example on how to use the UObject architecture as a middleware.

Part II — urbiscript User Manual

This part, also known as the “urbiscript tutorial”, teaches the reader how to program in urbiscript. It goes from the basis to concurrent and event-based programming. No specific knowledge is expected. There is no need for a C++ compiler, as `UObject` will not be covered here (see [Part I](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part IV](#)).

Chapter 8 — First Steps

First contacts with urbiscript.

Chapter 9 — Basic Objects, Value Model

A quick introduction to objects and values.

Chapter 10 — Flow Control Constructs

Basic control flow: `if`, `for` and the like.

Chapter 11 — Advanced Functions and Scoping

Details about functions, scopes, and lexical closures.

Chapter 12 — Objective Programming, urbiscript Object Model

A more in-depth introduction to object-oriented programming in urbiscript.

Chapter 13 — Functional Programming

Functions are first-class citizens.

Chapter 14 — Parallelism, Concurrent Flow Control

The urbiscript operators for concurrency, tags.

Chapter 15 — Event-based Programming

Support for event-driven concurrency in urbiscript.

Chapter 16 — Urbi for ROS Users

How to use ROS from Urbi, and vice-versa.

Part III — Guidelines and Cook Books

This part contains guides to some specific aspects of Urbi SDK.

Chapter 17 — Installation

Complete instructions on how to install Urbi SDK.

Chapter 18 — Frequently Asked Questions

Some answers to common questions.

Chapter 19 — Urbi Guideline

Based on our own experience, and code that users have submitted to us, we suggest a programming guideline for Urbi SDK.

Chapter 20 — Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2.

Chapter 21 — Building Urbi SDK

Building Urbi SDK from the sources. How to install it, how to check it and so forth.

Part IV — Urbi SDK Reference Manual

This part defines the specifications of the urbiscript language. It defines the expected behavior from the urbiscript interpreter, the standard library, and the SDK. It can be used to check whether some code is valid, or browse urbiscript or C++ API for a desired feature. Random reading can also provide you with advanced knowledge or subtleties about some urbiscript aspects.

This part is not an urbiscript tutorial; it is not structured in a progressive manner and is too detailed. Think of it as a dictionary: one does not learn a foreign language by reading a dictionary. For an urbiscript Tutorial, see [Part II](#).

This part does not aim at giving advanced programming techniques. Its only goal is to define the language and its libraries.

Chapter 22 — Programs

Presentation and usage of the different tools available with the Urbi framework related to urbiscript, such as the Urbi server, the command line client, `umake`, ...

Chapter 23 — urbiscript Language Reference Manual

Core constructs of the language and their behavior.

Chapter 24 — urbiscript Standard Library

Listing of all classes and methods provided in the standard library.

Chapter 25 — Communication with ROS

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, see <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

Chapter 26 — Gostai Standard Robotics API

Also known as “The Urbi Naming Standard”: naming conventions in for standard hardware/software devices and components implemented as UObject and the corresponding slots/events to access them.

Part V — Urbi Platforms

This part covers the specific features of Urbi for some of the platforms it was ported to. Environments not described in this part are covered in separate, stand-alone, documentations.

Chapter 27 — Aldebaran Nao

Nao is a humanoid robot Nao from Aldebaran Robotics.

Chapter 28 — Bioloid

Using the Bioloid robot construction kit with Urbi.

Chapter 29 — Mindstorms NXT

LEGO Mindstorms NXT is a programmable robotics kit released by Lego in July 2006, replacing the first-generation LEGO Mindstorms kit. The kit consists of 519 Technic pieces, 3 servo motors, 4 sensors (ultrasonic, sound, touch, and light), 7 connection cables, a USB interface cable, and the NXT Intelligent Brick.

Chapter 30 — Gostai Open Jazz

Gostai Open Jazz is an entirely programmable robot. It is shipped with the Urbi SDK to develop with Jazz.

Chapter 31 — Pioneer 3-DX

Pioneer 3-DX8 is an agile, versatile intelligent mobile robotic platform updated to carry loads more robustly and to traverse sills more surely with high-performance current management to provide power when it’s needed.

Chapter 32 — Segway RMP

The Segway Robotic Mobility Platform is a robotic platform based on the Segway Personal Transporter.

Chapter 33 — Spykee

The Spykee is a WiFi-enabled robot built by Meccano (known as Erector in the United States). It is equipped with a camera, speaker, microphone, and moves using two tracks.

Chapter 34 — Webots

Using Cyberbotics’ Webots simulation environment with Urbi.

Part VI — Tables and Indexes

This part contains material about the document itself.

Chapter 35 — Notations

Conventions used in the type-setting of this document.

Chapter 36 — Release Notes

Release notes of Urbi SDK.

Chapter 37 — Licenses

Licenses of components used in Urbi SDK.

Chapter 38 — Bibliography

References to other documents such as documentation, scientific papers, etc.

Chapter 39 — Glossary

Definition of the terms used in this document.

Chapter 40 — List of Tables

Index of all the tables: list of keywords, operators, etc.

Chapter 41 — List of Figures

Index of all the figures: snapshots, schema, etc.

Chapter 42 — List of Figures

An index of concepts, objects, and some selected routines.

1.6 Documentation

You may want to look at the documentation of the latest version, and visit <http://urbiforge.org>, the Urbi community web site, and its [forum](#).

This document and others are updated regularly on the Gostai Web site:

- <http://www.gostai.com/downloads/urbi/2.7.5/doc/>
All the documentations for Urbi 2.7.5.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-sdk.pdf>
Urbi SDK Documentation (this document) in PDF.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-sdk.htmldir/>
Urbi SDK Documentation (this document) in several HTML files.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-sdk-single.htmldir/>
Urbi SDK Documentation (this document) in a single large HTML file.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/sdk-remote.htmldir/>
The Doxygen documentation of UObject/SDK Remote for C++.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/sdk-remote-java.htmldir/>
The Doxygen documentation of UObject/SDK Remote for Java.
- <http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-naming.pdf>
The Gostai Standard Robotics API, i.e., Chapter 26 as a standalone document.

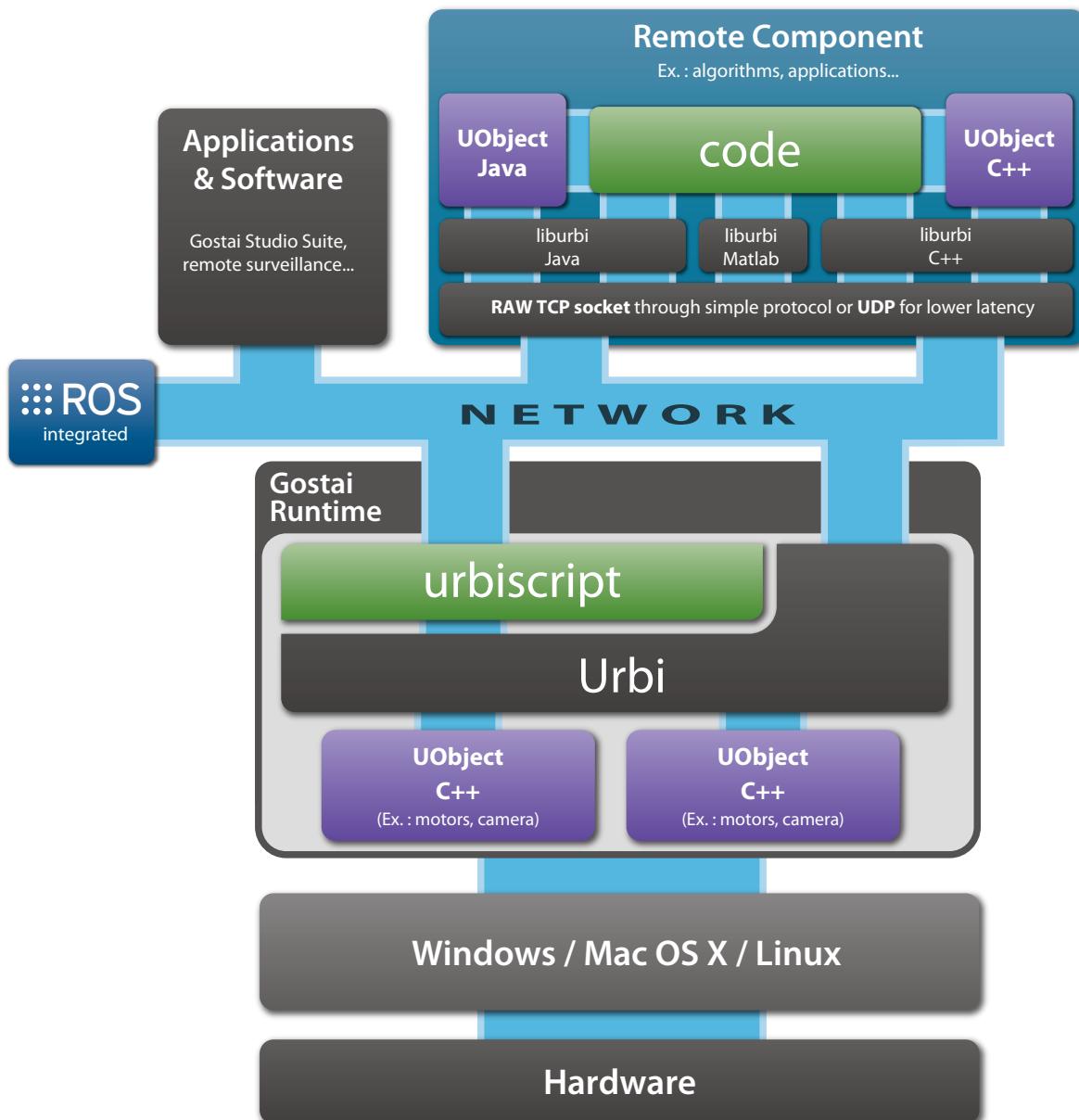


Figure 1.1: A Bird-View of the Urbi Architecture

Chapter 2

Contents

1	Introduction	3
1.1	Urbi and UObjects	3
1.2	The Big Picture	3
1.3	Urbi and urbiscript	4
1.4	Genesis	4
1.5	Outline	5
1.6	Documentation	8
2	Contents	11
3	Getting Started	35
I	Urbi and UObjects User Manual	39
4	Quick Start	43
4.1	UObject Basics	43
4.1.1	The Objects to Bind into Urbi	43
4.1.2	Wrapping into an UObject	44
4.1.3	Running Components	46
4.1.3.1	Compiling	46
4.1.3.2	Running UObjectS	47
4.2	Using urbiscript	48
4.2.1	The urbiscript Scripting Language	48
4.2.2	Concurrency	49
4.2.2.1	First Attempt	49
4.2.2.2	Second Attempt: Threaded Functions	50
4.3	Conclusion	50
5	The UObject API	53
5.1	Compiling UObjectS	53
5.1.1	Compiling by hand	53
5.1.2	The <code>umake-*</code> family of tools	54
5.1.3	Using the Visual C++ Wizard	54
5.2	Creating a class, binding variables and functions	55
5.3	Creating new instances	56
5.4	Binding functions	56
5.4.1	Simple binding	56
5.4.2	Multiple bindings	57
5.4.3	Asynchronous binding	57

5.5	Notification of a variable change or access	58
5.5.1	Threaded notification	58
5.6	Data-flow based programming: exchanging UVars	58
5.7	Data-flow based programming: InputPort	59
5.7.1	Customizing data-flow links	60
5.8	Timers	61
5.9	The special case of sensor/actuator variables	61
5.10	Using Urbi variables	61
5.11	Emitting events	62
5.12	UObject and Threads	62
5.13	Using binary types	62
5.13.1	UVar conversion and memory management	62
5.13.2	Binary conversion	62
5.13.3	0-copy mode	63
5.14	Direct Communication between UObject s	64
5.15	Using hubs to group objects	64
5.16	Sending urbiscript code	64
5.17	Using RTP transport in remote mode	65
5.17.1	Enabling RTP	65
5.17.2	Per-UVar control of RTP mode	65
5.18	Extending the cast system	65
5.18.1	Principle	65
5.18.2	Casting simple structures	66
6	The UObject Java API	69
6.1	Compiling and running UObject s	69
6.1.1	Compiling and running by hand	69
6.1.2	The <code>umake-java</code> and <code>urbi-launch-java</code> tools	70
6.2	Creating a class, binding variables and functions	70
6.3	Creating new instances	72
6.4	Binding functions	73
6.4.1	Simple binding	73
6.5	Notification of a variable change or access	74
6.6	Timers	74
6.7	Using Urbi variables	75
6.8	Sending Urbi code	75
6.9	Providing a main class or not	75
6.10	Import the examples with Eclipse	76
6.11	Run the UObject Java examples	78
7	Use Cases	81
7.1	Writing a Servomotor Device	81
7.1.1	Caching	83
7.1.2	Using Timers	84
7.2	Using Hubs to Group Objects	84
7.2.1	Alternate Implementation	86
7.3	Writing a Camera Device	86
7.3.1	Optimization in Plugin Mode	88
7.4	Writing a Speaker or Microphone Device	88
7.5	Writing a Softdevice: Ball Detection	88

II urbiscript User Manual	91
8 First Steps	95
8.1 Comments	95
8.2 Literal values	95
8.3 Function calls	96
8.4 Variables	97
8.5 Scopes	97
8.6 Method calls	97
8.7 Function definition	98
8.8 Conclusion	99
9 Basic Objects, Value Model	101
9.1 Objects in urbiscript	101
9.2 Methods	104
9.3 Everything is an object	105
9.4 The urbiscript values model	105
9.5 Conclusion	107
10 Flow Control Constructs	109
10.1 if	109
10.2 while	109
10.3 for	110
10.4 switch	110
10.5 do	110
11 Advanced Functions and Scoping	113
11.1 Scopes as expressions	113
11.2 Advanced scoping	113
11.3 Local functions	114
11.4 Lexical closures	114
12 Objective Programming, urbiscript Object Model	115
12.1 Prototype-Based Programming in urbiscript	115
12.2 Prototypes and Slot Lookup	116
12.3 Copy on Write	118
12.4 Defining Pseudo-Classes	118
12.5 Constructors	120
12.6 Operators	120
12.7 Properties	120
12.7.1 Features of Values	121
12.7.2 Features of Slots	121
13 Functional Programming	123
13.1 First class functions	123
13.2 Lambda functions	123
13.3 Lazy arguments	124
14 Parallelism, Concurrent Flow Control	127
14.1 Parallelism operators	127
14.2 Detach	128
14.3 Tags for parallel control flows	129
14.4 Advanced example with parallelism and tags	130

15 Event-based Programming	133
15.1 Watchdog constructs	133
15.2 Events	134
15.2.1 Emitting Events	134
15.2.2 Emitting events with a payload	135
16 Urbi for ROS Users	139
16.1 Communication on topics	139
16.1.1 Starting a process from Urbi	139
16.1.2 Listening to Topics	139
16.1.3 Advertising on Topics	140
16.1.3.1 Simple Talker	140
16.1.3.2 Turtle Simulation	141
16.2 Using Services	142
16.3 Image Publisher from ROS to Urbi	142
16.4 Image Subscriber from Urbi to ROS	144
16.5 Remote communication	145
III Guidelines and Cook Books	147
17 Installation	151
17.1 Download	151
17.2 Install & Check	151
17.2.1 GNU/Linux and Mac OS X	152
17.2.2 Windows	152
18 Frequently Asked Questions	153
18.1 Build Issues	153
18.1.1 Complaints about ‘+=’	153
18.1.2 error: ‘<anonymous>’ is used uninitialized in this function	153
18.1.3 AM_LANGINFO_CODESET	153
18.1.4 configure: error: The Java VM java failed	153
18.1.5 ‘make check’ fails	154
18.1.6 check: error: Unable to load native library: libjava.jnilib	154
18.2 Troubleshooting	154
18.2.1 error while loading shared libraries: libport.so	154
18.2.2 Error 1723: “A DLL required for this install to complete could not be run.”	154
18.2.3 When executing a program, the message “The system cannot execute the specified program.” is raised.	155
18.2.4 When executing a program, the message “This application has failed to start” is raised.	155
18.2.5 The server dies with “stack exhaustion”	155
18.2.6 ‘myuobject: file not found’. What can I do?	155
18.2.6.1 Getting a better diagnostic	155
18.2.6.2 GNU/Linux	155
18.2.6.3 Mac OS X	156
18.2.6.4 Windows	157
18.3 urbiscript	158
18.3.1 Objects lifetime	158
18.3.1.1 How do I create a new Object derivative?	158
18.3.1.2 How do I destroy an Object?	158
18.3.2 Slots and variables	159

18.3.2.1	Is the lobby a scope?	159
18.3.2.2	How do I add a new slot in an object?	162
18.3.2.3	How do I modify a slot of my object?	163
18.3.2.4	How do I create or modify a local variable?	163
18.3.2.5	How do I make a constructor?	163
18.3.2.6	How can I manipulate the list of prototypes of my objects?	164
18.3.2.7	How can I know the slots available for a given object?	164
18.3.2.8	How do I create a new function?	164
18.3.3	Tags	164
18.3.3.1	How do I create a tag?	164
18.3.3.2	How do I stop a tag?	164
18.3.3.3	Can tagged statements return a value?	165
18.3.4	Events	165
18.3.4.1	How do I create an event?	165
18.3.4.2	How do I emit an event?	165
18.3.4.3	How do I catch an event?	165
18.3.5	Standard Library	165
18.3.5.1	How can I iterate over a list?	165
18.4	UObjects	165
18.4.1	Is the UObject API Thread-Safe?	165
18.4.1.1	Plugin mode	166
18.4.1.2	Remote mode	166
18.5	Miscellaneous	167
18.5.1	What has changed since the latest release?	167
18.5.2	How can I contribute to the code?	167
18.5.3	How do I report a bug?	167
19	Urbi Guideline	169
19.1	urbiscript Programming Guideline	169
19.1.1	Prefer Expressions to Statements	169
19.1.2	Avoid return	169
20	Migration from urbiscript 1 to urbiscript 2	171
20.1	\$(Foo)	171
20.2	delete Foo	171
20.3	emit Foo	172
20.4	eval(Foo)	172
20.5	foreach	172
20.6	group	172
20.7	loopn	173
20.8	new Foo	173
20.9	self	173
20.10	stop Foo	173
20.11	# line	173
20.12	tag+end	173
21	Building Urbi SDK	175
21.1	Requirements	175
21.1.1	Bootstrap	175
21.1.2	Build	177
21.1.3	Check	179
21.2	Check out	179
21.3	Bootstrap	179

21.4 Configure	179
21.4.1 configuration options	180
21.4.2 Windows: Cygwin	180
21.4.3 Building For Windows	181
21.4.4 Building for Windows using Cygwin	181
21.5 Compile	181
21.6 Install	182
21.7 Relocatable	182
21.8 Run	182
21.9 Check	183
21.9.1 Lazy test suites	183
21.9.2 Partial test suite runs	184
IV Urbi SDK Reference Manual	185
22 Programs	189
22.1 Environment Variables	189
22.1.1 Search Path Variables	189
22.1.2 Environment Variables	189
22.2 Special Files	191
22.3 <code>urbi</code> — Running an Urbi Server	191
22.3.1 Options	191
22.3.2 Quitting	193
22.4 <code>urbi-image</code> — Querying Images from a Server	194
22.4.1 Options	194
22.5 <code>urbi-launch</code> — Running a UObject	194
22.5.1 Invoking <code>urbi-launch</code>	195
22.5.2 Examples	195
22.6 <code>urbi-launch-java</code> — Running a Java UObject	196
22.6.1 Invoking <code>urbi-launch-java</code>	196
22.7 <code>urbi-ping</code> — Checking the Delays with a Server	196
22.7.1 Options	196
22.8 <code>urbi-send</code> — Sending urbiscript Commands to a Server	197
22.9 <code>urbi-sound</code> — Querying Sounds from a Server	198
22.9.1 Options	198
22.10 <code>umake</code> — Compiling UObject Components	198
22.10.1 Invoking <code>umake</code>	198
22.10.2 <code>umake</code> Wrappers	200
23 urbiscript Language Reference Manual	201
23.1 Syntax	201
23.1.1 Characters, encoding	201
23.1.2 Comments	201
23.1.3 Synclines	202
23.1.4 Identifiers	202
23.1.5 Keywords	203
23.1.6 Literals	203
23.1.6.1 Angles	203
23.1.6.2 Dictionaries	203
23.1.6.3 Durations	203
23.1.6.4 Floats	203
23.1.6.5 Lists	205

23.1.6.6 Strings	206
23.1.6.7 Tuples	207
23.1.6.8 Pseudo classes	207
23.1.7 Statement Separators	209
23.1.7.1 ';'	209
23.1.7.2 ','	209
23.1.7.3 ' '	210
23.1.7.4 '&'	210
23.1.8 Operators	210
23.1.8.1 Arithmetic operators	211
23.1.8.2 Assignment operators	211
23.1.8.3 Postfix Operators	213
23.1.8.4 Bitwise operators	214
23.1.8.5 Logical operators	214
23.1.8.6 Comparison operators	215
23.1.8.7 Container operators	216
23.1.8.8 Object operators	217
23.1.8.9 All operators summary	217
23.2 Scopes and local variables	217
23.2.1 Scopes	217
23.2.2 Local variables	219
23.3 Functions	219
23.3.1 Function Definition	219
23.3.2 Arguments	221
23.3.3 Return value	221
23.3.4 Call messages	222
23.3.5 Strictness	222
23.3.6 Lexical closures	222
23.3.7 Variadic functions	223
23.4 Objects	224
23.4.1 Slots	224
23.4.1.1 Manipulation	224
23.4.1.2 Syntactic Sugar	225
23.4.2 Properties	225
23.4.2.1 Manipulation	225
23.4.2.2 Standard Properties	225
23.4.3 Prototypes	226
23.4.3.1 Manipulation	226
23.4.3.2 Inheritance	227
23.4.3.3 Copy on write	227
23.4.4 Sending messages	228
23.5 Enumeration types	228
23.6 Structural Pattern Matching	229
23.6.1 Basic Pattern Matching	230
23.6.2 Variable	230
23.6.3 Guard	231
23.7 Imperative flow control	231
23.7.1 break	231
23.7.2 continue	232
23.7.3 do	232
23.7.4 if	232
23.7.5 for	233
23.7.5.1 C-like for	233

23.7.5.2 Range-for	233
23.7.5.3 for n -times	234
23.7.6 if	234
23.7.7 loop	235
23.7.8 switch	235
23.7.9 while	236
23.7.9.1 while;	236
23.7.9.2 while—	237
23.8 Exceptions	237
23.8.1 Throwing exceptions	237
23.8.2 Catching exceptions	238
23.8.3 Inspecting exceptions	239
23.8.4 Finally	239
23.8.4.1 Regular execution	240
23.8.4.2 Control-flow	240
23.8.4.3 Exceptions	241
23.9 Assertions	242
23.9.1 Asserting an Expression	242
23.9.2 Assertion Blocks	243
23.10 Parallel and event-based flow control	243
23.10.1 at	243
23.10.1.1 at on Events	243
23.10.1.2 at on Boolean Expressions	244
23.10.1.3 Synchronous and asynchronous at	244
23.10.1.4 Scoping at at	245
23.10.2 every	245
23.10.2.1 every 	246
23.10.3 watch	247
23.10.3.1 every,	248
23.10.4 for	248
23.10.4.1 C-for,	248
23.10.4.2 range-for& (:).	249
23.10.4.3 for& (n)	249
23.10.5 loop,	250
23.10.6 waituntil	250
23.10.6.1 waituntil on Events	250
23.10.6.2 waituntil on Boolean Expressions	251
23.10.7 whenever	251
23.10.7.1 whenever on Events	252
23.10.7.2 whenever on Boolean Expressions	253
23.10.8 While	253
23.10.8.1 while,	253
23.11 Trajectories	254
23.12 Garbage collection and limitations	255
24 urbiscript Standard Library	257
24.1 Barrier	257
24.1.1 Prototypes	257
24.1.2 Construction	257
24.1.3 Slots	257
24.2 Binary	258
24.2.1 Prototypes	258
24.2.2 Construction	258

24.2.3 Slots	258
24.3 Boolean	259
24.3.1 Prototypes	259
24.3.2 Construction	259
24.3.3 Truth Values	260
24.3.4 Slots	260
24.4 CallMessage	260
24.4.1 Examples	260
24.4.1.1 Evaluating an argument several times	260
24.4.1.2 Strict Functions	261
24.4.2 Slots	261
24.5 Channel	263
24.5.1 Prototypes	263
24.5.2 Construction	264
24.5.3 Slots	264
24.6 Code	265
24.6.1 Prototypes	266
24.6.2 Construction	266
24.6.3 Slots	266
24.7 Comparable	267
24.7.1 Slots	268
24.8 Container	268
24.8.1 Prototypes	268
24.8.2 Slots	268
24.9 Control	269
24.9.1 Prototypes	269
24.9.2 Slots	269
24.10 Date	270
24.10.1 Prototypes	270
24.10.2 Construction	270
24.10.3 Slots	271
24.11 Dictionary	273
24.11.1 Example	274
24.11.2 Hash values	274
24.11.3 Prototypes	274
24.11.4 Construction	274
24.11.5 Slots	275
24.12 Directory	278
24.12.1 Prototypes	279
24.12.2 Construction	279
24.12.3 Slots	279
24.13 Duration	284
24.13.1 Prototypes	284
24.13.2 Construction	284
24.13.3 Slots	284
24.14 Enumeration	285
24.14.1 Prototypes	285
24.14.2 Examples	285
24.14.3 Construction	285
24.14.4 Slots	285
24.15 Event	287
24.15.1 Prototypes	287
24.15.2 Examples	287

24.15.3 Construction	287
24.15.4 Slots	287
24.16Exception	288
24.16.1 Prototypes	288
24.16.2 Construction	288
24.16.3 Slots	288
24.16.4 Specific Exceptions	289
24.17Executable	292
24.17.1 Prototypes	292
24.17.2 Construction	292
24.17.3 Slots	292
24.18File	292
24.18.1 Prototypes	292
24.18.2 Construction	292
24.18.3 Slots	293
24.19Finalizable	295
24.19.1 Example	295
24.19.2 Prototypes	296
24.19.3 Construction	296
24.19.4 Slots	297
24.20Float	297
24.20.1 Prototypes	297
24.20.2 Construction	297
24.20.3 Slots	298
24.21Float.limits	305
24.21.1 Slots	305
24.22FormatInfo	306
24.22.1 Prototypes	306
24.22.2 Construction	307
24.22.3 Slots	307
24.23Formatter	308
24.23.1 Prototypes	308
24.23.2 Construction	309
24.23.3 Slots	309
24.24Global	309
24.24.1 Prototypes	310
24.24.2 Slots	310
24.25Group	315
24.25.1 Example	315
24.25.2 Prototypes	316
24.25.3 Construction	316
24.25.4 Slots	316
24.26Hash	317
24.26.1 Prototypes	317
24.26.2 Construction	317
24.26.3 Slots	317
24.27InputStream	318
24.27.1 Prototypes	318
24.27.2 Construction	318
24.27.3 Slots	318
24.28IoService	319
24.28.1 Example	319
24.28.2 Prototypes	320

24.28.3 Construction	320
24.28.4 Slots	320
24.29 Job	321
24.29.1 Prototypes	321
24.29.2 Construction	321
24.29.3 Slots	321
24.30 Kernel1	322
24.30.1 Prototypes	322
24.30.2 Construction	323
24.30.3 Slots	323
24.31 Lazy	324
24.31.1 Examples	324
24.31.1.1 Evaluating once	324
24.31.1.2 Evaluating several times	325
24.31.2 Caching	325
24.31.3 Prototypes	326
24.31.4 Construction	326
24.31.5 Slots	326
24.32 List	326
24.32.1 Prototypes	327
24.32.2 Examples	327
24.32.3 Construction	327
24.32.4 Slots	327
24.33 Loadable	337
24.33.1 Prototypes	337
24.33.2 Example	337
24.33.3 Construction	338
24.33.4 Slots	338
24.34 Lobby	339
24.34.1 Prototypes	339
24.34.2 Construction	339
24.34.3 Examples	339
24.34.4 Slots	339
24.35 Location	343
24.35.1 Prototypes	343
24.35.2 Construction	343
24.35.3 Slots	344
24.36 Logger	345
24.36.1 Examples	345
24.36.2 Prototypes	345
24.36.3 Construction	346
24.36.4 Slots	346
24.37 Math	347
24.37.1 Prototypes	347
24.37.2 Construction	348
24.37.3 Slots	348
24.38 Mutex	350
24.38.1 Prototypes	350
24.38.2 Construction	350
24.38.3 Slots	350
24.39 nil	350
24.39.1 Prototypes	350
24.39.2 Construction	350

24.39.3 Slots	350
24.40 Object	351
24.40.1 Prototypes	351
24.40.2 Construction	351
24.40.3 Slots	351
24.41 Orderable	361
24.42 OutputStream	362
24.42.1 Prototypes	362
24.42.2 Construction	362
24.42.3 Slots	362
24.43 Pair	363
24.43.1 Prototype	363
24.43.2 Construction	363
24.43.3 Slots	363
24.44 Path	363
24.44.1 Prototypes	364
24.44.2 Construction	364
24.44.3 Slots	364
24.45 Pattern	366
24.45.1 Prototypes	366
24.45.2 Construction	366
24.45.3 Slots	366
24.46 Position	367
24.46.1 Prototypes	367
24.46.2 Construction	367
24.46.3 Slots	368
24.47 Primitive	369
24.47.1 Prototypes	369
24.47.2 Construction	369
24.47.3 Slots	369
24.48 Process	369
24.48.1 Prototypes	369
24.48.2 Example	370
24.48.3 Construction	370
24.48.4 Slots	370
24.49 Profile	372
24.49.1 Example	372
24.49.1.1 Basic profiling	372
24.49.1.2 Asynchronous profiling	373
24.49.2 Prototypes	375
24.49.3 Construction	375
24.49.4 Slots	375
24.50 Profile.Function	375
24.50.1 Prototypes	376
24.50.2 Construction	376
24.50.3 Slots	376
24.51 PseudoLazy	377
24.52 PubSub	377
24.52.1 Prototypes	377
24.52.2 Construction	377
24.52.3 Slots	377
24.53 PubSub.Subscriber	378
24.53.1 Prototypes	378

24.53.2 Construction	378
24.53.3 Slots	378
24.54 RangeIterable	378
24.54.1 Prototypes	378
24.54.2 Slots	378
24.55 Regexp	379
24.55.1 Prototypes	379
24.55.2 Construction	379
24.55.3 Slots	380
24.56 Semaphore	381
24.56.1 Prototypes	381
24.56.2 Construction	381
24.56.3 Slots	382
24.57 Server	383
24.57.1 Prototypes	383
24.57.2 Construction	383
24.57.3 Slots	383
24.58 Singleton	384
24.58.1 Prototypes	384
24.58.2 Construction	384
24.58.3 Slots	384
24.59 Socket	384
24.59.1 Example	385
24.59.2 Prototypes	386
24.59.3 Construction	386
24.59.4 Slots	386
24.60 StackFrame	387
24.60.1 Construction	388
24.60.2 Slots	388
24.61 Stream	388
24.61.1 Prototypes	388
24.61.2 Construction	388
24.61.3 Slots	388
24.62 String	389
24.62.1 Prototypes	389
24.62.2 Construction	389
24.62.3 Slots	389
24.63 System	394
24.63.1 Prototypes	394
24.63.2 Slots	394
24.64 System.PackageInfo	402
24.64.1 Prototypes	402
24.64.2 Slots	403
24.65 System.Platform	403
24.65.1 Prototypes	403
24.65.2 Slots	403
24.66 Tag	404
24.66.1 Examples	404
24.66.1.1 Stop	404
24.66.1.2 Block/unblock	405
24.66.1.3 Freeze/unfreeze	405
24.66.1.4 Scope tags	406
24.66.1.5 Enter/leave events	406

24.66.1.6 Begin/end	408
24.66.2 Construction	408
24.66.3 Slots	409
24.66.4 Hierarchical tags	410
24.67 Timeout	410
24.67.1 Prototypes	410
24.67.2 Construction	410
24.67.3 Examples	411
24.67.4 Slots	411
24.68 Traceable	411
24.68.1 Slots	411
24.69 TrajectoryGenerator	412
24.69.1 Prototypes	412
24.69.2 Examples	412
24.69.2.1 Accel	412
24.69.2.2 Cos	413
24.69.2.3 Sin	413
24.69.2.4 Smooth	413
24.69.2.5 Speed	414
24.69.2.6 Time	414
24.69.2.7 Trajectories and Tags	415
24.69.3 Construction	416
24.69.4 Slots	416
24.70 Triplet	416
24.70.1 Prototype	416
24.70.2 Construction	417
24.70.3 Slots	417
24.71 Tuple	417
24.71.1 Prototype	417
24.71.2 Construction	417
24.71.3 Slots	418
24.72 UObject	419
24.72.1 Prototypes	419
24.72.2 Slots	419
24.73 UObject	419
24.73.1 Prototypes	419
24.73.2 Slots	419
24.74 UValue	420
24.75 UVar	420
24.75.1 Construction	420
24.75.2 Prototypes	420
24.75.3 Slots	420
24.76 void	422
24.76.1 Prototypes	422
24.76.2 Construction	422
24.76.3 Slots	422
25 Communication with ROS	423
25.1 Ros	423
25.1.1 Construction	423
25.1.2 Slots	423
25.2 Ros.Topic	424
25.2.1 Construction	424

25.2.2 Slots	425
25.2.2.1 Common	425
25.2.2.2 Subscription	425
25.2.2.3 Advertising	425
25.2.3 Example	427
25.3 Ros.Service	428
25.3.1 Construction	428
25.3.2 Slots	428
26 Gostai Standard Robotics API	429
26.1 The Structure Tree	429
26.2 Frame of Reference	430
26.3 Component naming	431
26.4 Localization	431
26.5 Interface	433
26.5.1 AudioIn	433
26.5.2 AudioOut	434
26.5.3 Battery	434
26.5.4 BlobDetector	434
26.5.5 Identity	435
26.5.6 Led	435
26.5.6.1 RGBLed	435
26.5.7 Mobile	435
26.5.7.1 Blocking API	436
26.5.7.2 Speed-control API	436
26.5.7.3 Safety	436
26.5.7.4 State	437
26.5.8 Motor	437
26.5.8.1 LinearMotor	437
26.5.8.2 LinearSpeedMotor	437
26.5.8.3 RotationalMotor	438
26.5.8.4 RotationalSpeedMotor	438
26.5.9 Network	438
26.5.10 Sensor	438
26.5.10.1 AccelerationSensor	438
26.5.10.2 DistanceSensor	438
26.5.10.3 GyroSensor	439
26.5.10.4 Laser	439
26.5.10.5 TemperatureSensor	439
26.5.10.6 TouchSensor	439
26.5.11 SpeechRecognizer	440
26.5.12 TextToSpeech	440
26.5.13 Tracker	440
26.5.14 VideoIn	441
26.6 Standard Components	442
26.6.1 Yaw/Pitch/Roll orientation	442
26.6.2 Standard Component List	442
26.7 Compact notation	446
26.8 Support classes	446
26.8.1 Interface	447
26.8.2 Component	447
26.8.3 Localizer	447

V Urbi Platforms	449
27 Aldebaran Nao	453
27.1 Introduction	453
27.2 Starting up	453
27.3 Accessing joints	454
27.3.1 Advanced parameters	454
27.3.1.1 Trajectory generator period	454
27.3.1.2 Motor back-end method	454
27.3.1.3 Motor command debugging	454
27.4 Leds	454
27.5 Camera	455
27.5.1 Slots	455
27.6 Whole body motion	456
27.7 Other sensors	456
27.8 Interfacing with NaoQi	456
27.8.1 Accessing the NaoQi shared memory region	456
27.8.2 Accessing standard NaoQi modules	456
27.8.3 Binding new NaoQi modules in Urbi	457
27.8.4 Writing NaoQi modules in Urbi	457
27.8.5 More examples	458
28 Bioloid	463
28.1 Introduction	463
28.2 Installing Urbi for Bioloid	463
28.2.1 Flashing the firmware	463
28.2.2 Getting Urbi and Urbi for Bioloid	464
28.3 First steps	464
28.3.1 Starting up	464
28.3.2 Motor features	464
28.3.3 Sensor features	465
29 Mindstorms NXT	467
29.1 Launching Urbi for Mindstorms NXT	467
29.2 First steps with Urbi to control Mindstorms NXT	467
29.2.1 Make basic movements	467
29.2.2 Improving the movements	468
29.2.3 Reading sensors	468
29.2.4 Tagging commands	469
29.2.5 Playing sounds	469
29.2.6 Cyclic moves	469
29.2.7 Parallelism	470
29.2.8 Using functions	471
29.2.9 Loading files	471
29.2.10 Conclusion	471
29.3 Default layout reference	471
29.3.1 Motors	471
29.3.2 Sensors	471
29.3.2.1 Bumper	472
29.3.2.2 Sonar	472
29.3.2.3 Decibel	472
29.3.2.4 Light	472
29.3.3 Battery	472

29.3.4 Beeper	472
29.3.5 Command	473
29.4 How to make its own layout	473
29.4.1 Instantiating Motors	473
29.4.2 Instantiating sensors	474
29.4.3 Other devices	474
29.5 Available UObject Devices	474
29.5.1 Servo	475
29.5.2 UltraSonicSensor	475
29.5.3 SoundSensor	475
29.5.4 LightSensor	476
29.5.5 Switch	476
29.5.6 Battery	476
29.5.7 Beeper	477
29.5.8 Command	477
29.5.9 Instances	477
29.5.10 Groups	478
30 Gostai Open Jazz	479
30.1 Getting started	480
30.1.1 Documentation and Support	480
30.1.1.1 Jazz User Manual	480
30.1.1.2 Getting Support	480
30.1.2 Requirements	480
30.1.3 Embedded Software	480
30.1.4 Connecting Jazz to your Local Area Network	480
30.1.5 Connecting to Jazz from your Computer	480
30.1.5.1 Obtaining Jazz IP Address	480
30.1.5.2 Gostai Suite	481
30.1.5.3 Telnet Connection to Urbi on Jazz	481
30.1.5.4 Ssh Access to Jazz Filesystem	481
30.1.6 Jazz internal website	482
30.1.7 Uploading and loading your code on Jazz	483
30.1.7.1 The <i>User Services</i> web page	483
30.1.7.2 Uploading Files on Jazz Filesystem using scp	483
30.1.8 Updating Jazz Software	484
30.1.9 Restarting Jazz Software	485
30.1.10 Changing Software Profile	485
30.1.11 Using the Examples	485
30.1.11.1 urbiscript Examples	486
30.1.11.2 UObjects examples	486
30.2 urbiscript API	488
30.2.1 Eye leds	488
30.2.1.1 robot.body.head.eye	488
30.2.1.2 Example	488
30.2.2 Docking	488
30.2.3 Head Motors	488
30.2.3.1 robot.body.head	488
30.2.3.2 Example	489
30.2.4 Microphone	489
30.2.4.1 Quickly Record the Audio Stream	489
30.2.4.2 robot.body.head.micro	489
30.2.4.3 Example	490

30.2.5 Network	490
30.2.5.1 robot.network	490
30.2.5.2 Example	491
30.2.6 Playing Sound	491
30.2.6.1 Example	491
30.2.7 Text to Speech	492
30.2.7.1 Example	492
30.2.8 Video Camera	492
30.2.8.1 Visualize the Video Stream	492
30.2.8.2 robot.camera	492
30.2.8.3 Example	493
30.2.9 Movement	494
30.2.9.1 urbiscript interface	494
30.2.9.2 Example	495
30.2.10 Screen Display	495
30.2.10.1 From urbiscript	495
30.2.10.2 Command Line	496
30.2.11 Sonars	496
30.2.11.1 urbiscript Interface	496
30.2.11.2 Example	496
30.2.12 IRs	496
30.2.12.1 urbiscript Interface	497
30.2.12.2 Example	497
30.2.13 Laser	497
30.2.13.1 robot.body.laser	497
30.3 urbiscript Library	497
30.3.1 AlsaMicrophone	497
30.3.2 AlsaSpeaker	498
30.3.3 EchoCanceller	498
30.4 urbiscript Interfaces	499
30.4.0.1 AutomaticGainControl	499
30.4.0.2 Denoiser	500
30.5 Troubleshooting	500
30.5.1 Get Jazz Version	500
30.5.2 Access Jazz Logs	500
30.5.3 Debug your Urbi Code	500
30.5.3.1 Monitor urbiscript Job Execution	500
30.5.3.2 Monitor UObjects Execution Time	501
30.5.4 Enable Core Dump	502
30.5.5 Monitor Jazz Processor and Memory Load	502
30.5.6 Monitor Jazz Hard-Drive Space	502
30.5.7 I/O Issues when Powering Robot Off	502
30.6 ROS support	502
30.6.1 About ROS	502
30.6.2 ROS support in Jazz	502
30.6.3 Enabling and configuring ROS service	503
30.6.4 Predefined topics	503
30.6.5 Using the navigation stack with ROS	503
30.6.5.1 Creating a map of your environment	503
30.6.5.2 Autonomous navigation	504
30.6.5.3 Troubleshooting	504

31 Pioneer 3-DX	505
31.1 Getting started	505
31.1.1 Prerequisites	505
31.1.2 Installation	505
31.1.3 Running	505
31.2 How to use Pioneer 3-DX robot	505
31.2.1 P3dx	506
31.2.2 P3dx.body	506
31.2.3 P3dx.body.odometry	506
31.2.4 P3dx.body.sonar	507
31.2.5 P3dx.body.laser	508
31.2.6 P3dx.body.camera	508
31.2.7 P3dx.body.x	508
31.2.8 P3dx.body.yaw	508
31.2.9 P3dx.planner	509
31.2.10 P3dx.body.battery	509
31.3 Mobility modes	509
31.4 About units	509
32 Segway RMP	511
32.1 RMP	511
32.1.1 Instantiation	511
32.1.2 Slots	511
32.2 GSRAPI compliance	512
33 Spykee	513
33.1 Installing Urbi on the Spykee	513
33.2 Device list	514
33.3 Dynamically loaded modules	515
33.3.1 Clapper	515
34 Webots	517
34.1 Setup	517
34.1.1 Installation	517
34.1.1.1 Linux	517
34.1.1.2 Mac OS X	518
34.1.1.3 Windows	518
34.1.2 License	518
34.1.2.1 Evaluation mode	518
34.1.2.2 Setting up a license	518
34.2 Using Urbi as a controller in your Webots worlds	519
34.2.1 Using the default urbi-2.0 controller	519
34.2.2 Defining custom Urbi controllers	520
34.3 Binding your own robots with Urbi for Webots	520
34.4 Built-in robots and worlds	520
34.5 Webots UObjects	521
34.5.1 Robot devices UObjects	521
34.5.1.1 Accelerometer	521
34.5.1.2 Camera	521
34.5.1.3 Differential Wheels	522
34.5.1.4 Distance Sensor	522
34.5.1.5 Led	522
34.5.1.6 Servo	523

34.5.1.7 Touch Sensor	523
34.5.2 Supervisor API UObjects	523
34.5.2.1 Label	523
34.5.2.2 Manipulate Node	524
34.5.2.3 Simulation Controller	524
VI Tables and Indexes	527
35 Notations	531
35.1 Words	531
35.2 Frames	531
35.2.1 C++ Code	531
35.2.2 Grammar Excerpts	532
35.2.3 Java Code	532
35.2.4 Shell Sessions	533
35.2.5 urbiscript Sessions	533
35.2.6 urbiscript Assertions	533
36 Release Notes	535
36.1 Urbi SDK 2.7.5	535
36.1.1 Fixes	535
36.1.2 Changes	535
36.1.3 New Feature	536
36.1.4 Documentation	536
36.2 Urbi SDK 2.7.4	536
36.2.1 Fixes	536
36.2.2 Changes	537
36.2.3 Documentation	537
36.3 Urbi SDK 2.7.3	537
36.3.1 Fixes	537
36.3.2 Changes	538
36.3.3 Documentation	538
36.4 Urbi SDK 2.7.2	538
36.4.1 Fixes	538
36.5 Urbi SDK 2.7.1	538
36.5.1 Fixes	538
36.5.2 Changes	538
36.5.3 Documentation	538
36.6 Urbi SDK 2.7	539
36.6.1 Changes	539
36.6.2 New Features	539
36.6.3 Documentation	540
36.7 Urbi SDK 2.6	540
36.7.1 Fixes	540
36.7.2 Optimizations	540
36.7.3 New Features	541
36.7.4 Documentation	541
36.8 Urbi SDK 2.5	541
36.8.1 Fixes	541
36.8.2 New Features	541
36.8.3 Changes	542
36.8.4 Documentation	543

36.9 Urbi SDK 2.4	543
36.9.1 Fixes	543
36.9.2 New Features	544
36.9.3 Documentation	545
36.10 Urbi SDK 2.3	545
36.10.1 Fixes	545
36.10.2 New Features	545
36.11 Urbi SDK 2.2	546
36.11.1 Fixes	546
36.11.2 New Features	547
36.11.3 Documentation	548
36.12 Urbi SDK 2.1	548
36.12.1 Fixes	548
36.12.2 New Features	548
36.12.3 Optimization	549
36.12.4 Documentation	549
36.13 Urbi SDK 2.0.3	549
36.13.1 New Features	549
36.13.2 Fixes	550
36.13.3 Documentation	550
36.14 Urbi SDK 2.0.2	550
36.14.1 urbascript	550
36.14.2 Fixes	550
36.14.3 Documentation	550
36.15 Urbi SDK 2.0.1	550
36.15.1 urbascript	551
36.15.2 Documentation	551
36.15.3 Fixes	551
36.16 Urbi SDK 2.0	551
36.16.1 urbascript	551
36.16.1.1 Changes	551
36.16.1.2 New features	551
36.16.2 UObjects	552
36.16.3 Documentation	552
36.17 Urbi SDK 2.0 RC 4	552
36.17.1 urbascript	553
36.17.1.1 Changes	553
36.17.1.2 New objects	553
36.17.1.3 New features	553
36.17.2 UObjects	553
36.18 Urbi SDK 2.0 RC 3	553
36.18.1 urbascript	553
36.18.1.1 Fixes	553
36.18.1.2 Changes	553
36.18.2 Documentation	553
36.19 Urbi SDK 2.0 RC 2	554
36.19.1 Optimization	554
36.19.2 urbascript	554
36.19.2.1 New constructs	554
36.19.2.2 New objects	554
36.19.2.3 New features	555
36.19.2.4 Fixes	555
36.19.2.5 Deprecations	555

36.19.2.6 Changes	555
36.19.3 UObjects	556
36.19.4 Documentation	556
36.19.5 Various	557
36.20 Urbi SDK 2.0 RC 1	557
36.20.1 Auxiliary programs	557
36.20.2 urbiscript	558
36.20.2.1 Changes	558
36.20.2.2 Fixes	558
36.20.3 URBI Remote SDK	558
36.20.4 Documentation	558
36.21 Urbi SDK 2.0 beta 4	558
36.21.1 Documentation	558
36.21.2 urbiscript	558
36.21.2.1 Bug fixes	558
36.21.2.2 Changes	559
36.21.3 Programs	559
36.21.3.1 Environment variables	559
36.21.3.2 Scripting	559
36.21.3.3 urbi-console	559
36.21.3.4 Auxiliary programs	559
36.22 Urbi SDK 2.0 beta 3	560
36.22.1 Documentation	560
36.22.2 urbiscript	560
36.22.2.1 Fixes	560
36.22.2.2 Changes	560
36.22.3 UObjects	562
36.22.4 Auxiliary programs	562
36.23 Urbi SDK 2.0 beta 2	563
36.23.1 urbiscript	563
36.23.2 Standard library	563
36.23.3 UObjects	564
36.23.4 Run-time	564
36.23.5 Bug fixes	564
36.23.6 Auxiliary programs	565
37 Licenses	567
37.1 Boost Software License 1.0	567
37.2 BSD License	567
37.3 Expat License	568
37.4 gnu.bytecode	568
37.5 ICU License	569
37.6 Independent JPEG Group's Software License	570
37.7 Libcoroutine License	571
37.8 OpenSSL License	571
37.9 ROS	573
37.10 Urbi Open Source Contributor Agreement	575
38 Bibliography	577
39 Glossary	579
40 List of Tables	583

41 List of Figures	585
42 Index	587

Chapter 3

Getting Started

urbiscript comes with a set of tools, two of which being of particular importance:

urbi launches an Urbi server. There are several means to interact with it, which we will see later.

urbi-launch runs Urbi components, the UObjects, and connects them to an Urbi server.

Please, first make sure that these tools are properly installed. If you encounter problems, please see the frequently asked questions ([Chapter 18](#)), and the detailed installation instructions ([Chapter 17](#)).

```
# Make sure urbi is properly installed.  
$ urbi --version  
Urbi version 2.7.4 rev. 268868e  
Copyright (C) 2004-2012 Gostai S.A.S..  
  
Libport version urbi-sdk-2.7.4 rev. f870ce6  
Copyright (C) 2005-2012 Gostai S.A.S..
```

Shell Session

There are several means to interact with a server spawned by ***urbi***, see [Section 22.3](#) for details. First of all, you may use the options ‘**-e**/‘**--expression** *code*’ and ‘**-f**/‘**--file** *file*’ to send some *code* or the contents of some *file* to the newly run server. The option ‘**q**/‘**--quiet**’ discards the banner.

You may combine any number of these options, but beware that being event-driven, the server does not “know” when a program ends. Therefore, batch programs should end by calling **shutdown**. Using a Unix shell:

```
# A classical program.  
$ urbi -q -e 'echo("Hello, World!");' -e 'shutdown;'  
[00000004] *** Hello, World!
```

Shell Session

Listing 3.1: A batch session under Unix.

If you are running Windows, then, since the quotation rules differ, run:

```
# A classical program.  
$ urbi -q -e "echo("""Hello, World!""");" -e "shutdown;"  
[00000004] *** Hello, World!
```

Shell Session

Listing 3.2: A batch session under Windows.

To run an interactive session, use option ‘**-i**/‘**--interactive**’. Like most interactive interpreters, Urbi will evaluate the given commands and print out the results.

```
$ urbi -i  
[00000825] *** ****  
[00000825] *** Urbi version 2.7.4 rev. 268868e  
[00000825] *** Copyright (C) 2004-2012 Gostai S.A.S.
```

Shell Session

```
[00000825] ***
[00000825] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00000825] *** be used under certain conditions. Type 'license;',
[00000825] *** 'authors;', or 'copyright;' for more information.
[00000825] ***
[00000825] *** Check our community site: http://www.urbiforge.org.
[00000825] *** ****
1+2;
[00001200] 3
shutdown;
```

Listing 3.3: An interactive session under Unix.

The output from the server is prefixed by a number surrounded by square brackets: this is the date (in milliseconds since the server was launched) at which that line was sent by the server. This is useful at occasions, since Urbi is meant to run many parallel commands. But since these timestamps are irrelevant in most examples, they will often be filled with zeroes through this documentation.

Under Unix, the program `rlwrap` provides additional services (history of commands, advanced command line edition etc.); run '`rlwrap urbi -i`'.

In either case the server can also be made available for network-based interactions using option '`--port port`'. Note that while `shutdown` asks the server to quit, `quit` only quits one interactive session. In the following example (under Unix) the server is still available for other, possibly concurrent, sessions.

Shell Session	<pre>\$ urbi --port 54000 & [1] 77024 \$ telnet localhost 54000 Trying 127.0.0.1... Connected to localhost. Escape character is '^]'. [00004816] *** [00004816] *** Urbi version 2.7.4 rev. 268868e [00004816] *** Copyright (C) 2004-2012 Gostai S.A.S. [00004816] *** [00004816] *** This program comes with ABSOLUTELY NO WARRANTY. It can [00004816] *** be used under certain conditions. Type 'license;', [00004816] *** 'authors;', or 'copyright;' for more information. [00004816] *** [00004816] *** Check our community site: http://www.urbiforge.org. [00004816] *** 12345679*8; [00018032] 98765432 quit; Connection closed by foreign host.</pre>
---------------	---

Listing 3.4: An interactive session under Unix.

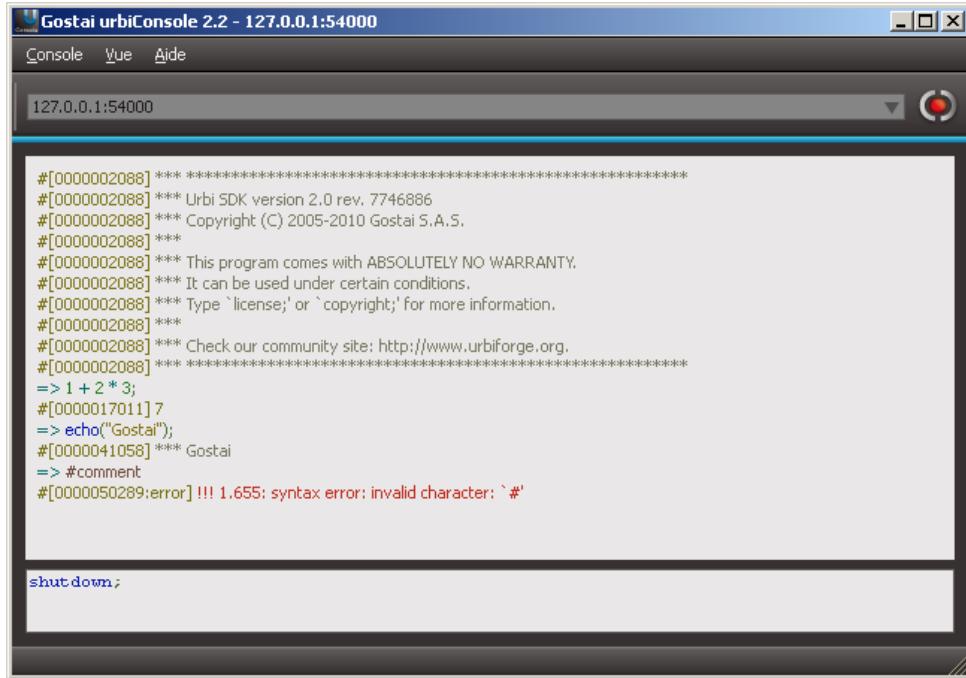
Under Windows, instead of using `telnet`, you may use `Gostai Console` (part of the package), which provides a Graphical User Interface to a network-connection to an Urbi server. To launch the server, run:

Shell Session	C:\> start urbi --port 54000
---------------	------------------------------

Listing 3.5: Starting an interactive session under Windows.

and to launch the client, click on `Gostai Console` which is installed by the installer.

Then, the interaction proceeds in the `Gostai Console` windows. Specify the host name and port to use ('`127.0.0.1:54000`') in the text field in the top of the window and click on the right to start the connection.



The program `urbi-send` (see [Section 22.8](#)) provides a nice interface to send batches of instructions (and/or files) to a running server.

```
$ urbi-send -P 54000 -e "1+2*3;" -Q
[00018032] 7
# Have the server shutdown;
$ urbi-send -P 54000 -e "shutdown;"
```

Shell Session

You can now send commands to your Urbi server. If at any point you get lost, or want a fresh start, you can simply close and reopen your connection to the server to get a clean environment. In some cases, particularly if you made global changes in the environment, it is simpler to start anew: shut your current server down using the command `shutdown`, and spawn a new one. In interactive mode you can also use the shortcut sequence Ctrl-D, like in many other interpreters.

In case of a foreground task preventing you to execute other commands, you can use Ctrl-C to kill the foreground task, clear queued commands and restore interactive mode.

You are now ready to proceed to the urbiscript tutorial: see [Part II](#).
Enjoy!

Part I

Urbi and UObjects User Manual

About This Part

This part covers the Urbi architecture: its core components (client/server architecture), how its middleware works, how to include extensions as UObjects (C++ components) and so forth.

No knowledge of the urbiscript language is needed. As a matter of fact, Urbi can be used as a standalone middleware architecture to orchestrate the execution of existing components.

Yet urbiscript is a feature that “comes for free”: it is easy using it to experiment, prototype, and even program fully-featured applications that orchestrate native components. The interested reader should read either the urbiscript user manual ([Part II](#)), or the reference manual ([Chapter 23](#)).

[Chapter 4 — Quick Start](#)

This chapter, self-contained, shows the potential of Urbi used as a middleware.

[Chapter 5 — The UObject API](#)

This section shows the various steps of writing an Urbi C++ component using the UObject API.

[Chapter 6 — The UObject Java API](#)

UObjects can also be written in Java. This section demonstrates it all.

[Chapter 7 — Use Cases](#)

Interfacing a servomotor device as an example on how to use the UObject architecture as a middleware.

Chapter 4

Quick Start

This chapter presents Urbi SDK with a specific focus on its middleware features. It is self-contained in order to help readers quickly grasp the potential of Urbi used as a middleware. References to other sections of this document are liberally provided to point the reader to the more complete documentation; they should be ignored during the first reading.

4.1 UObject Basics

As a simple running example, consider a (very) basic factory. Raw material delivered to the factory is pushed into some assembly machines, which takes some time.

4.1.1 The Objects to Bind into Urbi

As a first component of this factory, the core machine of the factory is implemented as follows. This class is pure regular C++, it uses no Urbi feature at all.

The header ‘`machine.hh`’, which declares `Machine`, is traditional. The documentation uses the Doxygen syntax.

```
#ifndef MACHINE_MACHINE_HH
#define MACHINE_MACHINE_HH

#include <urbi/uobject.hh>

class Machine
{
public:
    /// Construction.
    /// \param duration how long the assembly process takes.
    ///                 In seconds.
    Machine(float duration);

    /// Lists of strings.
    typedef std::list<std::string> strings;

    /// Assemble the raw components into a product.
    std::string operator()(const strings& components) const;

    /// The duration of the assembly process, in seconds.
    /// Must be positive.
    float duration;
};

#endif // ! MACHINE_MACHINE_HH
```

The implementation file, ‘`machine.cc`’, is equally straightforward.

```
// A wrapper around Boost.Foreach.
```

```
#include <libport/foreach.hh>

#include "machine.hh"

Machine::Machine(float d)
    : duration(d)
{
    assert(0 <= d);
}

std::string
Machine::operator()(const std::vector<std::string>& components) const
{
    // Waiting for duration seconds.
    useconds_t one_second = 1000 * 1000;
    usleep(useconds_t(duration * one_second));

    // Iterate over the list of strings (using Boost.Foreach), and
    // concatenate them in res.
    std::string res;
    foreach (const std::string& s, components)
        res += s;
    return res;
}
```

4.1.2 Wrapping into an UObject

By *binding* a UObject, we mean using the UObject API to declare objects to the Urbi world. These objects have member variables (also known as *attributes*) and/or member functions (also known as *methods*) all of them or some of them being declared to Urbi.

One could modify the Machine class to make it a UObject, yet we recommend wrapping pure C++ classes into a different, wrapping, UObject. It is strongly suggested to aggregate the native C++ objects in the UObject — rather than trying to derive from it. By convention, we prepend a “U” to the name of the base class, hence the UMachine class. This class provides a simplified interface, basically restricted to what will be exposed to Urbi. It must derive from urbi::UObject.

```
#ifndef MACHINE_UMACHINE_HH
#define MACHINE_UMACHINE_HH

// Include the UObject declarations.
#include <urbi/uobject.hh>

// We wrap factories.
#include "machine.hh"

/// A UObject wrapping a machine object.
class UMachine
    : public urbi::UObject
{
public:
    /// C++ constructor.
    /// \param name name given to the instance.
    UMachine(const std::string& name);

    /// Urbi constructor.
    /// \param d the duration of the assembly process.
    /// Must be positive.
    /// \return 0 on success.
    int init(ufloat d);

    /// Wrapper around Machine::operator().
}
```

```

    std::string assemble(std::list<std::string> components);

    /// Function notified when the duration is changed.
    /// \param v the UVar being modified (i.e., UMachine::duration).
    /// \return 0 on success.
    int duration_set(urbi::UVar& v);

private:
    /// The duration of the assembly process.
    urbi::UVar duration;

    /// The actual machine, wrapped in this UObject.
    Machine* machine;
};

#endif // ! MACHINE_UMACHINE_HH

```

The implementation of `UMachine` is simple. It uses some of the primitives used in the binding process ([Section 5.2](#)):

- `UStart(class)` declares classes that are `UObjects`; eventually such classes will appear in `urbiscript` as `uobjects.class` (but since `Global` derives from `uobjects` you may just use the simpler name `class`). Use it once.
- `UBindFunction(class, function)` declares a *function*. Eventually bound in the `urbiscript` world as `uobjects.class.function`.
- Similarly, `UBindVar(class, variable)` declares a *variable*.

Urbi relies on the prototype model for object-oriented programming, which is somewhat different from the traditional C++ class-based model ([Chapter 9](#)). This is reflected by the presence of *two* different constructors:

- `UMachine::UMachine`, the C++ constructor invoked for every single instance of the `UObject`. It is always invoked by the Urbi system when instantiating a `UObject`, *including* the prototype itself. Its sole argument is its name (an internal detail you need not be aware of). The C++ constructor must register the `UMachine::init` function. It may also bind functions and variables.
- `UMachine::init`, the Urbi constructor invoked each time a new clone of `UMachine` is made, i.e., for every instance except the first one.

Functions and variables that do not make sense for the initial prototype (which might not be fully functional) should be bound here, rather than in the C++ constructor.

The following listing is abundantly commented, and is easy to grasp.

```

#include "umachine.hh"

// Register the UMachine UObject in the Urbi world.
UStart(UMachine);

// Bouncing the name to the UObject constructor is mandatory.
UMachine::UMachine(const std::string& name)
    : urbi::UObject(name)
    , machine(0)
{
    // Register the Urbi constructor. This is the only mandatory
    // part of the C++ constructor.
    UBindFunction(UMachine, init);
}

int
UMachine::init(ufloat d)

```

```

{
    // Failure on invalid arguments.
    if (d < 0)
        return 1;

    // Bind the functions, i.e., declare them to the Urbi world.
    UBindFunction(UMachine, assemble);
    UBindThreadedFunctionRename
        (UMachine, assemble, "threadedAssemble", urbi::LOCK_FUNCTION);
    // Bind the UVars before using them.
    UBindVar(UMachine, duration);

    // Set the duration.
    duration = d;
    // Build the machine.
    machine = new Machine(d);

    // Request that duration_set be invoked each time duration is
    // changed. Declared after the above "duration = d" since we don't
    // want it to be triggered for this first assignment.
    UNotifyChange(duration, &UMachine::duration_set);

    // Success.
    return 0;
}

int
UMachine::duration_set(urbi::UVar& v)
{
    assert(machine);
    ufloat d = static_cast<ufloat>(v);
    if (0 <= d)
    {
        // Valid value.
        machine->duration = d;
        return 0;
    }
    else
        // Report failure.
        return 1;
}

std::string
UMachine::assemble(std::list<std::string> components)
{
    assert(machine);

    // Bounce to Machine::operator().
    return (*machine)(components);
}

```

4.1.3 Running Components

As a first benefit from using the Urbi environment, this source code is already runnable! No `main` function is needed, the Urbi system provides one.

4.1.3.1 Compiling

Of course, beforehand, we need to compile this UObject into some loadable module. The Urbi modules are *shared objects*, i.e., libraries that can be loaded on demand (and unloaded) during the execution of the program. Their typical file names depend on the architecture: ‘`machine.so`’

on most Unix platforms (including Mac OS X), and ‘`machine.dll`’ on Windows. To abstract away from these differences, we will simply use the base name, ‘`machine`’ with the Urbi tool chain.

There are several options to compile our machine as a UObject. If you are using Microsoft Visual Studio, the Urbi SDK installer created a “UObject” project template accessible through the “New project” wizard. Otherwise, you can use directly your regular compiler tool chain. You may also use `umake-shared` from the ‘`umake-*`’ family of programs ([Section 22.10.2](#)):

```
$ ls machine.uob
machine.cc machine.hh umachine.cc umachine.hh
$ umake-shared machine.uob -o machine
# ... Lots of compilation log messages ...
$ ls
_ubuild-machine.so machine.la machine.so machine.uob
```

Shell Session

The various files are:

‘`machine.uob`’ Merely by convention, the sources of our UObject are in a ‘`*.uob`’ directory.

‘`machine.so`’ A shared dlopen-module. This is the “true” product of the `umake-shared` invocation. Its default name can be quite complex (‘`uobject-i386-apple-darwin9.7.0.so`’ on my machine), as it will encode information about the architecture of your machine; if you don’t need such accuracy, use the option ‘`--output`/‘`-o`’ to specify the output file name.

`umake-shared` traversed ‘`machine.uob`’ to gather and process relevant files (source files, headers, libraries and object files) in order to produce this output file.

‘`_ubuild-machine.so`’ this is a temporary directory in which the compilation takes place. It can be safely removed by hand, or using `umake-deepclean` ([Section 22.10.2](#)).

‘`machine.la`’ a [GNU Libtool](#) file that contains information such as dependencies on other libraries. While this file should be useless most of the time, we recommend against removing it: it may help understand some problems.

4.1.3.2 Running UObject

There are several means to toy with this simple UObject. You can use `urbi-launch` ([Section 22.5](#)) to plug the UMachine in an Urbi server and enter an interactive session.

```
# Launch an Urbi server with UMachine plugged in.
$ urbi-launch --start machine -- --interactive
[00000000] *** *****
[00000000] *** Urbi SDK version 2.7 rev. a6a1ec5
[00000000] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00000000] ***
[00000000] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00000000] *** be used under certain conditions. Type 'license;',
[00000000] *** 'authors;', or 'copyright;' for more information.
[00000000] ***
[00000000] *** Check our community site: http://www.urbiforge.org.
[00000000] ***
var f = UMachine.new(1s);
[00020853] UMachine_0x1899c90
f.assemble(["Hello, ", "World!"]);
[00038705] "Hello, World!"
shutdown;
```

Shell Session

You may also launch the machine UObject in the background, as a network component:

```
$ urbi-launch --start machine --host 0.0.0.0 --port 54000 &
```

Shell Session

and interact with it using your favorite client (`telnet`, `netcat`, `socat`, ...), or using the `urbi-send` tool ([Section 22.8](#)).

Shell Session

```
$ urbi-send --port 54000          \
-e 'UMachine.assemble([12, 34]);' \
--quit
[00126148] "1234"
[00000000:client_error] End of file
$ urbi-send --port 54000          \
-e 'var f = UMachine.new(1s)|' \
-e 'f.assemble(["ab", "cd"]);' \
--quit
[00146159] "abcd"
[00000000:client_error] End of file
```

4.2 Using urbiscript

urbiscript is a programming language primarily designed for robotics. Its syntax is inspired by that of C++: if you know C, C++, Java or C#, writing urbiscript programs is easy. It's a dynamic object-oriented ([Chapter 12](#)) scripting language, which makes it well suited for high-level application. It supports and emphasizes parallel ([Chapter 14](#)) and event-based programming ([Chapter 15](#)), which are very popular paradigms in robotics, by providing core primitives and language constructs.

Thanks to its client/server approach, one can easily interact with a robot, to monitor it, or to experiment live changes in the urbiscript programs.

Courtesy of the UObject architecture, urbiscript is fully integrated with C++. As already seen in the above examples ([Section 4.1.3](#)), you can bind C++ classes in urbiscript seamlessly. urbiscript is also integrated with many other languages such as Java ([Chapter 6](#)), MatLab or Python. The UObject framework also naturally provides urbiscript with support for distributed architectures: objects can run in different processes, possibly on remote computers.

4.2.1 The urbiscript Scripting Language

The following example shows how one can easily interface UObject into the urbiscript language. The following simple class (actually, a genuine object, in urbiscript “classes are objects”, see [Chapter 12](#)) aggregates two assembly machines, a fast one, and a slow one. This class demonstrates usual object-oriented, sequential, features.

urbiscript Session

```
class TwoMachineFactory
{
    // A shorthand common to all the Two Machine factories.
    var UMachine = uobjects.UMachine;

    // Default machines.
    var fastMachine = UMachine.new(10ms);
    var slowMachine = UMachine.new(100ms);

    // The urbiscript constructor.
    // Build two machines, a fast one, and a slow one.
    function init(fast = 10ms, slow = 100ms)
    {
        // Make sure fast <= slow.
        if (slow < fast)
            [fast, slow] = [slow, fast];
        // Two machines for each instance of TwoMachineFactory.
        fastMachine = UMachine.new(fast);
        slowMachine = UMachine.new(slow);
    };
}
```

```
// Wrappers to make invocation of the machine simpler.
function fast(input) { fastMachine.assemble(input); }
function slow(input) { slowMachine.assemble(input); }

// Use the slow machine for large jobs.
function assemble(input)
{
    var res|
    var machine|
    if (5 < input.size)
        { machine = "slow" | res = slow(input); }
    else
        { machine = "fast" | res = fast(input); } |
    echo ("Used the %s machine (%s => %s)" % [machine, input, res]) |
    res
};

[00000001] TwoMachineFactory
```

Using this class is straightforward.

```
var f = TwoMachineFactory.new|;
f.assemble([1, 2, 3, 4, 5, 6]);
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
[00000003] "123456"
f.assemble([1]);
[00000004] *** Used the fast machine ([1] => 1)
[00000005] "1"
```

urbiscript
Session

The genuine power of urbiscript is when concurrency comes into play.

4.2.2 Concurrency

Why should we wait for the slow job to finish if we have a fast machine available? To do so, we must stop requesting a *sequential* composition between both calls. We did that by using the sequential operator, ‘;’. In urbiscript, there exists its concurrent counter-part: ‘,’. Indeed, running *a, b* means “launch the program *a* and then launch the program *b*”. The urbiscript Manual contains a whole section devoted to explaining these operators ([Chapter 14](#)).

4.2.2.1 First Attempt

Let's try it:

```
f.assemble([1, 2, 3, 4, 5, 6]),
f.assemble([1]),
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
[00000004] *** Used the fast machine ([1] => 1)
```

urbiscript
Session

This is a complete failure.

Why?

Since Urbi cannot expect that your code is thread-safe, by default all calls to UObject features are *synchronous*, or *blocking*: *the whole Urbi system is suspended until the function returns*. There is a single thread of execution for Urbi, and when calling a function from a (plugged) UObject, that thread of execution is devoted to evaluated the code.

See for instance below that, in even though `f.assemble` is slow and launched in background, the almost instantaneous display of `ping` is not performed immediately.

```
echo(f.slow([1, 2, 3, 4, 5, 6])),
echo("ping");
[00000002] *** 123456
[00000002] *** ping
```

urbiscript
Session

There are several means to address this unintended behavior. If the base library provides a threaded API (in our example, the `Machine` class, not the `UMachine` UObject wrapper), then you could use it. Yet we don't recommend it, as it takes away from Urbi the possibility to really control concurrency issues (for instance it cannot turn non-blocking functions into blocking functions).

A better option is to ask Urbi to turn blocking function calls into non-blocking ones.

4.2.2.2 Second Attempt: Threaded Functions

If you read carefully the body of the `UMachine::init` function, you will find the following piece of code:

`C++`

```
UBindFunction(UMachine, assemble);
UBindThreadedFunctionRename
    (UMachine, assemble, "threadedAssemble", urbi::LOCK_FUNCTION);
```

Both calls bind the function `UMachine::assemble`, but the second one will run the call in a separate thread. Since multiple calls to a single function (or different functions of a single object etc.) are likely to fail, a locking model must be provided. Here, `urbi::LOCK_FUNCTION` means that concurrent calls to `UMachine::assemble` must be serialized: one at a time.

To use this threaded version of `assemble`, we can simply patch our `TwoMachineFactory` class:

`urbiscript Session`

```
do (TwoMachineFactory)
{
    fast = function (input) { fastMachine.threadedAssemble(input) };
    slow = function (input) { slowMachine.threadedAssemble(input) };
};
```

Let's try again where we failed previously ([Section 4.2.2.1](#)):

`urbiscript Session`

```
f.assemble([1, 2, 3, 4, 5, 6]),
f.assemble([1]),
[00000004] *** Used the fast machine ([1] => 1)
[00000002] *** Used the slow machine ([1, 2, 3, 4, 5, 6] => 123456)
sleep(200ms);
```

Victory! The fast machine answered first.

You may have noticed that the result is no longer reported. Indeed, the `urbiscript` interactive shell only displays the result of synchronous expressions (i.e., those ending with a ‘;’): asynchronous answers are confusing (see the inversion here).

`Channels` are useful to “send” asynchronous answers.

`urbiscript Session`

```
var c1 = Channel.new("c1")|;
var c2 = Channel.new("c2")|;
c1 << f.assemble([10, 20, 30, 40, 50, 60]),
c2 << f.assemble([100]),
sleep(200ms);
[00000535] *** Used the fast machine ([100] => 100)
[00000535:c2] "100"
[00000625] *** Used the slow machine ([10, 20, 30, 40, 50, 60] => 102030405060)
[00000625:c1] "102030405060"
```

4.3 Conclusion

This section gave only a quick glance to all the power that Urbi provides to support concurrency. Actually, it makes plenty of sense to embed an Urbi engine in a native C++ program, and to delegate the concurrency issues to it. Thanks to its middleware and client/server architecture, it is then possible to connect it to remote components of different kinds, such as using ROS for instance.

Then, a host of features are at hand, ready to be used when you need them: event-driven programming, automatic monitoring of expressions, interactive sessions, ... and, last but not least, the urbiscript programming language.

Chapter 5

The UObject API

The UObject API can be used to add new objects written in C++ to the urbiscript language, and to interact from C++ with the objects that are already defined. We cover the use cases of controlling a physical device (servomotor, speaker, camera...), and interfacing higher-lever components (voice recognition, object detection...) with Urbi.

The C++ API defines the UObject class. To each instance of a C++ class deriving from UObject will correspond an urbiscript object sharing some of its methods and attributes. The API provides methods to declare which elements of your object are to be shared. To share a variable with Urbi, you have to give it the type UVar. This type is a container that provides conversion and assignment operators for all types known to Urbi: `double`, `std::string` and `char*`, and the binary-holding structures `UBinary`, `USound` and `UIImage`. This type can also read from and write to the liburbi `UValue` class. The API provides methods to set up callbacks functions that will be notified when a variable is modified or read from Urbi code. Instance methods of any prototype can be rendered accessible from urbiscript, providing all the parameters types and the return type can be converted to/from `UValue`.

5.1 Compiling UObject

UObjects can be compiled easily directly with any regular compiler. Nevertheless, Urbi SDK provides two tools to compile UObject seamlessly.

In the following sections, we will try to compile a shared library named ‘factory.so’ (or ‘factory.dll’ on Windows platforms) from a set of four files (‘factory.hh’, ‘factory.cc’, ‘ufactory.hh’, ‘ufactory.cc’). These files are stored in a ‘factory.uob’ directory; its name bares no importance, yet the ‘*.uob’ extension makes clear that it is a UObject.

In what follows, *urbi-root* denotes the top-level directory of your Urbi SDK package, see [Section 17.2](#).

5.1.1 Compiling by hand

On Unix platforms, compiling by hand into a shared library is straightforward:

```
$ g++ -I urbi-root/include \
    -fPIC -shared \
    factory.uob/*cc -o factory.so
$ file factory.so
factory.so: ELF 32-bit LSB shared object, Intel 80386, \
    version 1 (SYSV), dynamically linked, not stripped
```

Shell Session

On Mac OS X the flags ‘`-Wl,-undefined,dynamic_lookup`’ are needed:

```
$ g++ -I urbi-root/include \
    -shared -Wl,-undefined,dynamic_lookup \
    factory.uob/*.cc -o factory.so
$ file factory.so
```

Shell Session

```
factory.so: Mach-O 64-bit dynamically linked shared library x86_64
```

5.1.2 The umake-* family of tools

`umake` can be used to compile UObjects. See [Section 22.10](#) for its documentation.

You can give it a list of files to compile:

Shell Session

```
$ umake -q --shared-library factory.uob/*.cc -o factory.so
umake: running to build library.
```

or directories in which C++ sources are looked for:

Shell Session

```
$ umake -q --shared-library factory.uob -o factory.so
umake: running to build library.
```

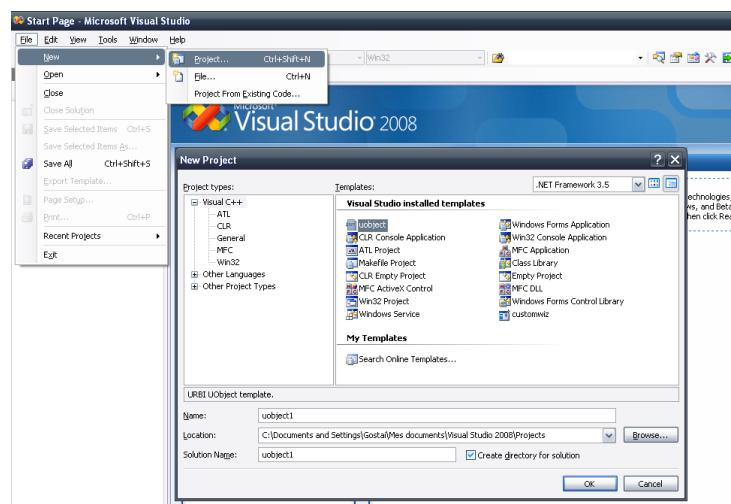
or finally, if you give no argument at all, the sources in the current directory:

Shell Session

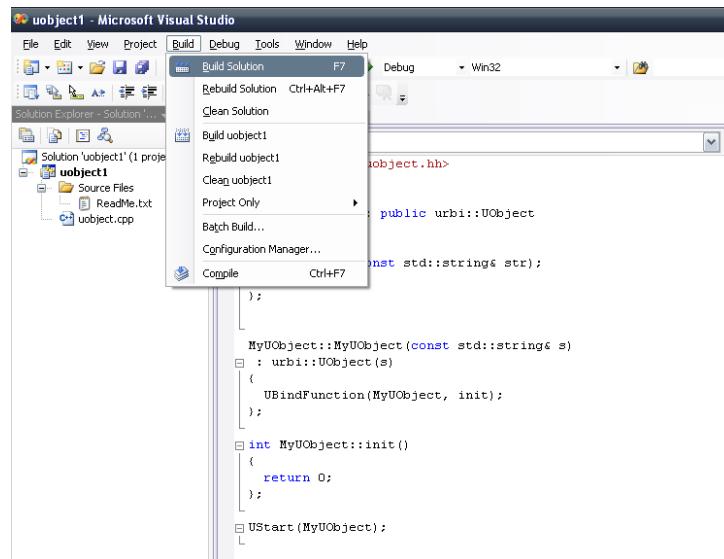
```
$ cd factory.uob
$ umake -q --shared-library -o factory.so
umake: running to build library.
```

5.1.3 Using the Visual C++ Wizard

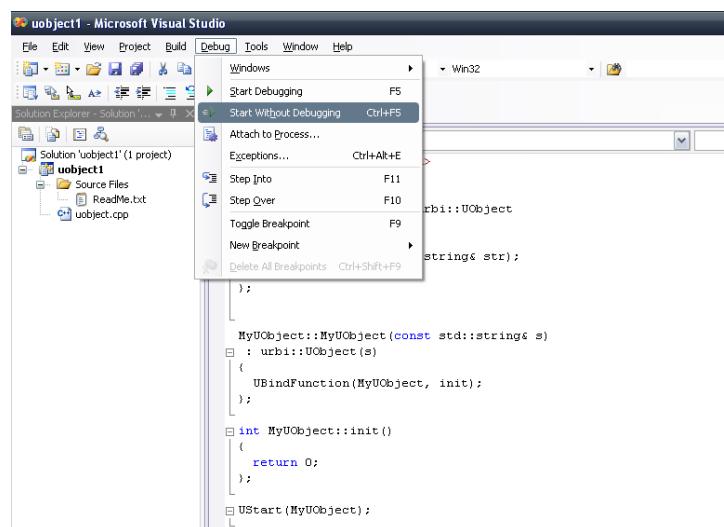
If you installed Urbi SDK using its installer, and if you had Visual C++ installed, then the UObject wizard was installed. Use it to create your UObject code:



Then, compile your UObject.



And run it.



5.2 Creating a class, binding variables and functions

Let's illustrate those concepts by defining a simple object: **adder**. This object has one variable **v**, and a method **add** that returns the sum of this variable and its argument.

- First the required include:

```
#include <urbi/uobject.hh>
```

C++

- Then we declare our **adder** class:

```

class adder : public urbi::UObject // Must inherit from UObject.
{
public:
    // The class must have a single constructor taking a string.
    adder(const std::string&);

    // Our variable.
    urbi::UVar v;

    // Our method.
    double add(double rhs) const;
};
```

C++

- The implementation of the constructor.

C++

```
// the constructor defines what is available from Urbi
adder::adder(const std::string& s)
    : urbi::UObject(s) // required
{
    // Bind the variable.
    UBindVar(adder, v);

    // Bind the function.
    UBindFunction(adder, add);
}
```

- The implementation of our `add` method.

C++

```
double
adder::add(double rhs) const
{
    return ((double) v) + rhs;
}
```

- And register this class:

C++

```
// Register the class to the Urbi kernel.
UStart(adder);
```

To summarize:

- Declare your object class as inheriting from `urbi::UObject`.
- Declare a single constructor taking a string, and pass this string to the constructor of `urbi::UObject`.
- Declare the variables you want to share with Urbi with the type `urbi::UVar`.
- In the constructor, use the macros `UBindVar(class-name, variable-name)` for each `UVar` you want as an instance variable, and `UBindFunction(class-name, function-name)` for each function you want to bind.
- Call the macro `UStart` for each object.

5.3 Creating new instances

When you start an Urbi server, an object of each class registered with `UStart` is created with the same name as the class. New instances can be created from Urbi using the `new` method. For each instance created in Urbi, a corresponding instance of the C++ object is created. You can get the arguments passed to the constructor by defining and binding a method named `init` with the appropriate number of arguments.

5.4 Binding functions

5.4.1 Simple binding

You can register any member function of your `UObject` using the macro `UBindFunction(class-name, function-name)`.

Once done, the function can be called from urbiscript.

The following types for arguments and return value are supported:

- Basic integer and floating types (int, double, float...).
- `const std::string&` or `const char*`.
- `urbi::UValue` or any of its subtypes (`UBinary`, `UList`...).
- `std::list`, `std::vector` or `boost::unordered_map` of the above types.
- `UObject*`. Just pass the `UObject` from urbiscript.
- `UVar&`. Pass `myObject.&myUvar` from urbiscript.

The procedure to register new types to this system is explained in [Section 5.18](#).

5.4.2 Multiple bindings

If you have multiple functions to bind, you can use the `UBindFunctions` macro to bind multiple functions at once:

`UBindFunctions(class-name, function1, function2...).`

5.4.3 Asynchronous binding

Functions bound using `UBindFunction` are called synchronously, and thus block everything until they return.

If you wish to bind a function that requires a non-negligible amount of time to execute, you can have it execute in a separate thread by calling

`UBindThreadedFunction(class-name, function-name, lock-mode).`

The function code will be executed in a separate thread without breaking the urbiscript execution semantics.

The `lock-mode` argument can be used to prevent parallel execution of multiple bound functions if your code is not thread-safe. It can be any of the following values.

- `LOCK_NONE`
No locking is performed.
- `LOCK_FUNCTION`
Parallel execution is limited to one instance of the bound function.
- `LOCK_FUNCTION_DROP`
Same as `LOCK_FUNCTION`, but operations are dropped instead of being queued if one is already running.
- `LOCK_FUNCTION_KEEP_ONE`
Same as `LOCK_FUNCTION`, but the queue is limited to one, and subsequent calls are dropped.
- `LOCK_INSTANCE`
Parallel execution is limited to one bound function for each object instance.
- `LOCK_CLASS`
Parallel execution is limited to one bound function for the class.
- `LOCK_MODULE`
Parallel execution is limited to one bound function for the whole module (shared object).

Other queue sizes can be used by passing `LockSpec(LOCK_FUNCTION, my-queue-size)` as `lock-mode`.

There is a restriction to the locking mechanism: *you cannot mix multiple locking modes*. For instance a function bound with `LOCK_FUNCTION` mode will not prevent another function bound with `LOCK_INSTANCE` from executing in parallel.

You can perform your own locking using semaphores if your code needs a more complex locking model.

You can limit the maximum number of threads that can run in parallel by using the `setThreadLimit` function.

5.5 Notification of a variable change or access

You can register a function that will be called each time a variable is modified by calling `UNotifyChange(var, func)`.

- `var` must be either the name of an `UVar`, or an `UVar` itself.
- `func` must be a member function of your `UObject`. This function will be called each time the `UVar` receives a new value.

The function can take 0 or 1 argument. If the argument is of type `UVar&`, then the function will receive the `UVar` that was passed to `UNotifyChange`. If it is of any other type, then the new value in the `UVar` will be converted to this type and passed to the function.

In plugin mode, there is a similar mechanism to create a *getter function* that will be called each time an `UVar` is accessed: the `UNotifyAccess` function. It has the same signature as `UNotifyChange`, and calls the given function each time someone tries to access the `UVar`. The function can update the value in the `UVar` before the access takes place. Usage of `UNotifyAccess` should be reserved to infrequently used `UVar` that take a long time to update, as it disrupts the data flow between `UObject`.

You can remove all notifies associated to any given `UVar` by calling its `unnotify` function.

5.5.1 Threaded notification

In a manner similar to `UBindThreadedFunction`, you can request your callback function to be called in a separate thread by using `UNotifyThreadedChange(var, func, lock-mode)`.

The `lock-mode` argument has the same semantic as for bound functions.

There is one restriction: the callback function must not take a `UVar&` as argument. This restriction is here to ensure that each invocation of your callback will receive the correct value that the source `UVar` had at call time.

5.6 Data-flow based programming: exchanging UVars

The `UNotifyChange` and `UNotifyAccess` features can be used to link multiple `UObjects` together, and perform data-flow based programming: the `UNotifyChange` can be called to monitor `UVars` from other `UObjects`. Those `UVars` can be transmitted through bound function calls.

One possible pattern is to have each data-processing `UObject` take its input from monitored `UVars`, given in its constructor, and output the result of its processing in other `UVars`. Consider the following example of an object-tracker:

```
C++
class ObjectTracker: public urbi:: UObject
{
public:
    ObjectTracker(const std::string& n)
        : urbi:: UObject(n)
    {
        // Bind our constructor.
        UBindFunction(ObjectTracker, init);
    }
    // Take our data source in our constructor.
    void init(UVar& image)
    {
```

```

UNotifyChange(image, &ObjectTracker::onImage);
// Bind our output variable.
UBindVar(ObjectTracker, val);
}
void onImage(UVar& src)
{
    UBinary b = src;
    // Processing here.
    val = processing_result;
}
UVar val;
};
UStart(ObjectTracker);

```

The following urbiscript code would be used to initialize an ObjectTracker given a camera:

```
var tracker = ObjectTracker.new(camera.&val);
```

urbiscript
Session

An other component could then take the tracker output as its input.

Using this model, chains of processing elements can be created. Each time the UObject at the start of the chain updates, all the notifyChange will be called synchronously in cascade to update the state of the intermediate components.

5.7 Data-flow based programming: InputPort

Urbi provides a second and more standard way to perform data-flow programming. In this approach, inputs of a component are declared as local InputPort, and the binding between this InputPort and the output of another component is done in urbiscript using the `>>` operator between two UVar:

```

class ObjectTracker: public urbi::UObject
{
    ObjectTracker(const std::string& n)
        : urbi::UObject(n)
    {
        // Bind our constructor.
        UBindFunction(ObjectTracker, init);
        // Bind our input port.
        UBindVar(ObjectTracker, input);
        // NotifyChange on our own input port
        UNotifyChange(input, &ObjectTracker::onImage);
    }

    // Init is empty.
    void init()
    {
    }

    // onImage is unchanged.
    void onImage(UVar& src)
    {
        UBinary b = src;
        // Processing here.
        val = processing_result;
    }

    UVar val;

    // Declare our input port.
    InputPort input;
};

UStart(ObjectTracker);

```

C++

In this model, linking the components is done in urbiscript:

urbiscript
Session

```
var tracker = ObjectTracker.new();
camera.&val >> tracker.&input;
```

5.7.1 Customizing data-flow links

The `>>` operator to establish a data-flow link between two `UVar` returns an object of type `UConnection` that can be used to customize the link.

This object is also present in the slot `changeConnections` of the source `UVar`.

Here is the list of `UConnection` slots:

- **`enabled`**
Set to false to disable the link.
- **`minInterval`**
Minimal interval in seconds at which the link can activate. Changes of the source at a higher rate will be ignored.
- **`asynchronous`**
If true, notifies on the target InputPort will trigger asynchronously. The system will also prevent two instances from running in parallel by dropping updates until the callback functions terminate.
- **`disconnect`**
Disconnect the link.
- **`reconnect(src)`**
Reconnect the link by changing the source to `src`.
- **`callCount`**
Number of times the link was reset.
- **`fireRate`**
Rate in Hertz at which the link triggers.
- **`meanCallTime`**
Average time taken by the callback function on the target InputPort.
- **`minCallTime`**
Minimum call time.
- **`maxCallTime`**
Maximum call time.
- **`resetStats`**
Reset all statistics.
- **`getAll`**
Return the list of all the connections in the system.

The function `uobjects.connectionStats()` can be used to display the statistics of all the connections, and `uobjects.resetConnectionStats()` can be called to reset all statistics.

5.8 Timers

The API provides two methods to have a function called periodically:

- `void urbi::UObject::USetUpdate(ufloat period)`
Set up a timer that calls the virtual method `UObject::update()` with the specified period (in milliseconds). Disable updates if `period` is -1.
- `urbi::TimerHandle urbi::UObject::USetTimer<T>(ufloat period, void (T::*fun)())`
Invoke an `UObject` member function `fun` every `period` milliseconds. `fun` is a regular member-function pointer, for instance `MyUObject::my_function`. The function returns a `TimerHandle` that can be passed to the `UObject::removeTimer(h)` function to disable the timer.

5.9 The special case of sensor/actuator variables

In Urbi, a variable can have a different meaning depending on whether you are reading or writing it: you can use the same variable to represent the target value of an actuator and the current value measured by an associated sensor. This special mode is activated by the `UObject` defining the variable by calling `UOwned` after calling `UBindVar`. This call has the following effects:

- When Urbi code or code in other modules read the variable, they read the current value.
- When Urbi code or code in other modules write the variable, they set the target value.
- When the module that called `UOwned` reads the variable, it reads the target value. When it writes the variable, it writes the current value.

5.10 Using Urbi variables

The C++ class `UVar` is used to represent any Urbi slot in C++. To bind the `UVar` to a specific slot, pass its name to the `UVar` constructor, or its `init` method. Once the `UVar` is bound, you can write any compatible type to it, and the new value will be visible in urbiscript. Similarly, you can cast the `UVar` (or use the `as()` method) to convert the current urbiscript value held to any compatible type.

Compatible types are the same as for bound functions (see [Section 5.18](#)).

```
// Set the camera format to 0 if it is 1.
UVar v;
v.init("camera", "format");
if (v == 1)
    v = 0;
```

C++

Some care must be taken in remote mode: changes on the variable coming from Urbi code or an other module can take time to propagate to the `UVar`. By default, all changes to the value will be sent to the remote `UObject`. To have more control on the bandwidth used, you can disable the automatic update by calling `unnotify()`. Then you can get the value on demand by calling `UVar::syncValue()`.

```
UVar v("Global", "x");
send("every|(100ms) Global.x = time,");
// At this point, v is updated approximately every 100 milliseconds.

v.unnotify();
// At this point v is no longer updated. If v was the only UVar pointing to
// 'Global.x', the value is no longer transmitted.

v.syncValue();
```

C++

```
// The previous call will ask for the value of Global.x once, and block until
// the value is written to v.
```

You can read and write all the Urbi properties of an **UVar** by reading and writing the appropriate **UProp** object in the **UVar**.

5.11 Emitting events

The **UEvent** class can be used to create and emit urbiscript events. Instances are created and initialized exactly as **UVar**: either by using the **UBindEvent** macro, or by calling one of its constructors or the **init** function.

Once initialized, the **emit** function will trigger the emission of the associated urbiscript event. It can be called with any number of arguments, of any compatible type.

5.12 UObject and Threads

The **UObject** API is thread-safe in both plugin and remote mode: All API calls including operations on **UVar** can be performed from any thread.

5.13 Using binary types

Urbi can store binary objects of any type in a generic container, and provides specific structures for sound and images. The generic containers is called **UBinary** and is defined in the ‘urbi/ubinary.hh’ header. It contains an enum field **type** giving the type of the binary (**UNKNOWN**, **SOUND** or **IMAGE**), and an union of a **USound** and **UIImage** struct containing a pointer to the data, the size of the data and type-specific meta-information.

5.13.1 UVar conversion and memory management

The **UBinary** manages its memory: when destroyed (or going out-of-scope), it frees all its allocated data. The **USound** and **UIImage** do not.

Reading an **UBinary** from a **UVar**, and writing a **UBinary**, **USound** or **UIImage** to an **UVar** performs a deep-copy of the data (by default, see below).

Reading a **USound** or **UIImage** from an **UVar** directly will perform a shallow copy from the internal data. The structure content is only guaranteed to be valid until the function returns, and should not be modified.

5.13.2 Binary conversion

To convert between various sound and image formats, two functions are provided in the header ‘urbi/uconversion.hh’:

C++

```
void urbi::convert(UIImage& source, UIImage& destination);
void urbi::convert(USound& source, USound& destination);
```

For those functions to work, destination must be filled correctly:

- data and size can be both 0, in which case data will be allocated for you using **malloc**. If data is set but size is too small to fit the value, data will be reallocated using **realloc**.
- all the description fields must be set. It is possible to set any field to 0, in which case the value from **source** will be used.

Consider this example of a sound algorithm requiring 8-bit mono input:

C++

```

class SoundAlgorithm: public UObject
{
public:
<...>
void init();
void onData(UVar& v);
// We reuse the same USound for converted data to avoid reallocation.
USound convertedData;
}

void SoundAlgorithm::init(UVar& dataSource)
{
    // initialize convertedData
convertedData.data = 0;
convertedData.size = 0; // Let convert allocate for us
convertedData.soundFormat = SOUND_RAW;
convertedData.channels = 1;
convertedData.rate = 0; // Use sample rate of the source
convertedData.sampleSize = 8;
convertedData.sampleFormat = SAMPLE_UNSIGNED;
UNotifyChange(dataSource, &SoundAlgorithm::onData);
}

void SoundAlgorithm::onData(UVar& v)
{
    USound src = v;
    convert(src, convertedData);
    // Work on convertedData.
}

```

5.13.3 0-copy mode

In plugin mode, you can setup any `UVar` in 0-copy mode by calling `setBypass(true)`. In this mode, binary data written to the `UVar` is not copied, but a reference is kept. As a consequence, the data is only available from within registered `notifyChange` callbacks. Those callbacks can use `UVar::val()` or cast the `UVar` to a `UBinary` & to retrieve the reference. Attempts to read the `UVar` from outside `notifyChange` will block until the `UVar` is updated again, and copy the value at this time.

An example will certainly clarify: Let us first declare an `UObject` that will generate binary data using 0-copy mode.

```

// Declare an UObject producing images in 0-copy optimized mode.
class OptimizedImageSource: public UObject
{
<...>
public:
    UVar val;
    UBinary imageData;
};

void OptimizedImageSource::init()
{
    // Bind val
    UBindVar(OptimizedImageSource, val);
    // Mark it as bypass mode
    val.setBypass(true);
    // Start a timer.
    USetUpdate(10);
}

int OptimizedImageSource::update()
{

```

C++

```

<Update imageData here>
// Notify all notifyChange callbacks without copying the data.
val = imageData;
}

```

Let us then declare an other component that will access this binary data without any copy:

C++

```

class BinaryProcessor: public UObject
{
public:
void init();
void onData(UVar& v);
InputPort binaryIn;
};

void BinaryProcessor::init()
{
UBindVar(BinaryProcessor, binaryIn);
UNotifyChange(binaryIn, &BinaryProcessor::onData);
}

void BinaryProcessor::onData(UVar& v)
{
const UBinary& b = v;
// If in urbiscript you connect the two components using:
// OptimizedImageSource.&val >> BinaryProcessor.&binaryIn
// then b will be OptimizedImageSource.binaryData, not a copy.
}

```

Typing `OptimizedImageSource.val` in urbiscript will wait for the next update from `OptimizedImageSource::update` and copy the data at this point.

5.14 Direct Communication between UObject

For modularity reasons, all interactions between `UObject`s should go through the various middleware communication mechanisms, mainly `InputPort` and `UNotifyChange`. But it is possible to access directly the C++ instance of an `UObject`:

- by binding a function taking a `UObject*` as argument, and calling it from urbiscript, passing the `UObject` itself.
- by calling the C++ function `getUObject(name)`. `name` must be the canonical `UObject` name, passed to your constructor, and stored in the `__name` member.

5.15 Using hubs to group objects

Sometimes, you need to perform actions for a group of `UObject`s, for instance devices that need to be updated together. The API provides the `UObjectHub` class for this purpose. To create a hub, simply declare a subclass of `UObjectHub`, and register it by calling once the macro `UStartHub` (*class-name*). A single instance of this class will then be created upon server start-up. `UObject` instances can then register to this hub by calling `URegister` (*hub-class-name*). Timers can be attached to `UObjectHub` the same way as to `UObject` (see [Section 5.8](#)). A hub instance can be retrieved by calling `getUObjectHub` (*string class-name*). The hub also holds the list of registered `UObject` in its `members` attribute.

5.16 Sending urbiscript code

If you need to send urbiscript code to the server, the `URBI()` macro is available, as well as the `send()` function. You can either pass it a string, or directly Urbi code inside a double pair of

parentheses:

```
send ("myTag:1+1;");

URBI (( at (myEvent?(var x)) { myTag:echo x; }; ));
```

urbiscript
Session

You can also use the `call` method to make an urbiscript function call:

```
// C++ equivalent of urbiscript 'System.someFunc(12, "foo");'
call("System", "someFunc", 12, "foo");
```

urbiscript
Session

5.17 Using RTP transport in remote mode

By default, Urbi uses TCP connections for all communications between the engine and remote UObjects. Urbi also supports the UDP-based *RTP* protocol for more efficient transmission of updated variable values. RTP will provide a lower latency at the cost of possible packet loss, especially in bad wireless network conditions.

5.17.1 Enabling RTP

To enable RTP connections, both the engine and the remote-mode urbi-launch containing your remote UObject must load the RTP UObject. This can be achieved by passing `urbi/rtp` as an extra argument to both urbi-launch command lines (one for the engine, the other for your remote UObject).

Once done, all binary data transfer (like sound and image) in both directions will by default use a RTP connection.

5.17.2 Per-UVar control of RTP mode

You can control whether a specific UVar uses RTP mode by calling its `useRTP(bool)` function. Each binary-type UVar will have its own RTP connection, and all non-binary UVar will share one.

From urbiscript, you can also write to the `rtp` slot of each UVar. Existing notifies will be modified to use rtp if you set it to true.

5.18 Extending the cast system

5.18.1 Principle

The same cast system is used both for bound function's arguments and return values, and for reading/writing UVar.

Should you want to add new type `MyType` to the system you must define two functions:

```
namespace urbi
{
    void operator, (UValue& v, const MyType& t)
    {
        // Here you must fill v with the serialized representation of t.
    }

    template<> struct uvalue_caster<MyType>
    {
        MyType operator()(UValue& v)
        {
            // Here you must return a MyType made with the information in v.
        }
    }
}
```

C++

Once done, you will be able without any other change to

- Take MyType as argument to a bound function.
- Return a MyType from a bound function.
- Write a MyType to an UVar.
- Convert an UVar to MyType using the UVar::as function.

5.18.2 Casting simple structures

The system provides facilities to serialize simple structures by value between C++ and urbiscript. This system uses two declarations of each structure, one in C++ and the other in Urbi, and maps between the two.

Here is a complete commented example to map a simple Point structure between urbiscript and C++.

C++

```
struct Point
{
    // Your struct must have a default constructor.
    Point()
        : x(0), y(0)
    {}
    double x, y;
};

// Declare the structure to the cast system. First argument is the struct,
// following arguments are the field names.
URBI_REGISTER_STRUCT(Point, x, y);
```

urbiscript
Session

```
// Declare the urbiscript structure. It must be globally accessible, and
// inheriting from UValueSerializable.
class Global.Point: UValueSerializable
{
    var x = 0;
    var y = 0;

    function init(var xx = 0, var yy = 0)
    {
        x = xx;
        y = yy
    };

    function asString()
    {
        "%s, %s" % [x, y]
    };
};
// Add the class to Serializables to register it.
var Serializables.Point = Point;
```

Once done, you can call bound functions taking a C++ Point by passing them an urbiscript Point and exchange Point between both worlds through UVar read/write:

C++

```
// This function can be bound using UBindFunction.
Point MyObject::opposite(Point p)
{
    return Point(-p.x, -p.y);
}

// Writing a Point to an UVar is OK.
void MyObject::writePoint(Point p)
```

```
{  
    UVar v(this, "val");  
    v = p;  
}  
  
// Converting an UVar to a Point is easy.  
ufloat MyObject::xCoord()  
{  
    UVar v(this, "val");  
    Point p;  
    // Fill p with content of v.  
    v.fill(p);  
    // Alternate for the above.  
    p = v.as<Point>();  
    return v.x;  
}
```


Chapter 6

The UObject Java API

The UObject Java API can be used to add new remote objects written in Java to the urbiscript language, and to interact from Java with the objects that are already defined. We cover the use cases of interfacing higher-level components (voice recognition, object detection...) with Urbi using Java.

The Java API defines the UObject class. To each instance of a Java class deriving from UObject will correspond an urbiscript object sharing some of its methods and attributes. The API provides methods to declare which elements of your object are to be shared. To share a variable with Urbi, you have to give it the type UVar. This type is a container that provides conversion and setter member functions for all types known to Urbi: `double`, `java.lang.String`, the binary-holding structures `urbi.UBinary`, `urbi.USound` and `urbi.UIImage`, list types `urbi.UList` and dictionaries `urbi.Dictionary`. This type can also read from and write to the `urbi.UValue` class. The API provides methods to set up callbacks functions that will be notified when a variable is modified or read from urbiscript code. Instance methods of any prototype can be made accessible from urbiscript, providing all the parameter types and the return type can be converted to/from `urbi.UValue`.

The UObject Java API has the following limitations:

- it is available only to create remote UObject, i.e., these objects run as separate processes.
- the Java library is generated from the C++ SDK implementation, and rely on compiled C++ code. Thus, remote Java UObject can only run on computers having the full Urbi native SDK installed.

6.1 Compiling and running UObject

UObjects can be compiled easily directly with the `javac` compiler, then you can create JAR archives using the `jar` tool.

In the following sections, we will try to create an uobject jar archive named ‘`machine.jar`’ from a set of two files (‘`Machine.java`’, ‘`UMachine.java`’).

In what follows, `urbi-root` denotes the top-level directory of your Urbi SDK package, see [Section 17.2](#).

6.1.1 Compiling and running by hand

To compile your UObject you need to include in the classpath `liburbijava.jar`:

```
$ javac -cp urbi-root/share/sdk-remote/java/lib/liburbijava.jar:. \
  Machine.java UMachine.java
$ jar -cvf machine.jar UMachine.class Machine.class
added manifest
adding: UMachine.class
adding: Machine.class
```

Shell
Session

Then to run your uobject, you need to call `java`. We provide a main class called `urbi.UMain` in the `liburbijava.jar` archive. You can use this class to start your UObjects. This class takes the names of your uobjects jar files as argument. You also need to specify the lib folder of the urbi SDK into `java.library.path`:

```
Shell Session $ java -Djava.library.path=urbi-root/lib
               -cp urbi-root/share/sdk-remote/java/lib/liburbijava.jar \
               urbi.UMain ./machine.jar
urbi-launch: obeying to URBI_ROOT = /usr/local/gostai
UObject: Urbi SDK version 2.3 rev. 3e93ec1
Copyright (C) 2004-2010 Gostai S.A.S.

Libport version urbi-sdk-2.3 rev. 66cb0ec
Copyright (C) 2004-2010 Gostai S.A.S.
UObject: Remote Component Running on 127.0.0.1 54000
Kernel Version: 0
[LibUObject] Registering function UMachine.init 1 into UMachine.init from UMachine
[LibUObject] Pushing UMachine.init in function
```

6.1.2 The `umake-java` and `urbi-launch-java` tools

`umake-java` can be used to compile Java UObjects. It will produce a JAR archive that you can use with `urbi-launch-java`.

You can give it a list of files to compile:

```
Shell Session $ umake-java -q machine.uob/*.java -o machine.jar
```

or directories in which C++ sources are looked for:

```
Shell Session $ umake-java -q machine.uob -o machine.jar
```

or finally, if you give no argument at all, the sources in the current directory:

```
Shell Session $ cd machine.uob
$ umake-java -q -o machine.jar
```

To run your UObject then use `urbi-launch-java` (see [Section 22.6](#)):

```
Shell Session $ urbi-launch-java machine.jar
urbi-launch: obeying to URBI_ROOT = /usr/local/gostai
UObject: Urbi SDK version 2.3 rev. 3e93ec1
Copyright (C) 2004-2010 Gostai S.A.S.

Libport version urbi-sdk-2.3 rev. 66cb0ec
Copyright (C) 2004-2010 Gostai S.A.S.
UObject: Remote Component Running on 127.0.0.1 54000
Kernel Version: 0
[LibUObject] Registering function UMachine.init 1 into UMachine.init from UMachine
[LibUObject] Pushing UMachine.init in function
```

6.2 Creating a class, binding variables and functions

Let's illustrate those concepts by defining a simple object: `adder`. This object has one variable `v`, and a method `add` that returns the sum of this variable and its argument.

- First you need some imports:

```
Java import urbi.UObject;
import urbi.UVar;
import urbi.UValue;
```

- Then we declare and implement our `adder` class:

```
public class Adder extends UObject // must extends UObject
{
    /// Register the class within urbi
    static { UStart(Adder.class); }

    /// Declare a variable v that will be accessible in Urbi
    private UVar v = new UVar();

    /// the class must have a single constructor taking a string
    public Adder (String s)
    {
        super (s);
        /// Bind the variable v to Urbi
        UBindVar (v, "v");

        /// Initialize our UVar v to some value
        /// (we choose 42 :)
        v.setValue(42);

        /// Bind the function add to Urbi
        UBindFunction ("add");
    }

    /// Our method.
    public double add (double rhs)
    {
        /// Return the value of our UVar v (converted to double)
        /// plus the value of the argument of the function.
        return v.doubleValue () + rhs;
    }
}
```

Java

To bind the variables to Urbi, we use the function:

```
void UBindVar (UVar v, String name)
```

Java

This function take as parameter the UVar variables, and the name of the UVar (because Urbi need to know what is the name of your variable). Once your variable is bound with `UBindVar` it will be accessible in Urbi.

- Each UObject needs to be registered within urbi using the code

```
static { UStart(YourUObject.class); }
```

Java

If you run this UObject and test it from Urbi it gives:

```
[00000102] *** *****
[00000102] *** Urbi version 2.0 rev. 96a4b2f
[00000102] *** Copyright (C) 2004-2010 Gostai S.A.S.
[00000102] ***
[00000102] *** This program comes with ABSOLUTELY NO WARRANTY.
[00000102] *** It can be used under certain conditions.
[00000102] *** Type 'license;' or 'copyright;' for more information.
[00000102] ***
[00000102] *** Check our community site: http://www.urbiforge.org.
[00000102] *** *****
Adder;
[00006783] Adder
Adder.v;
[00010871] 42
Adder.add(-26);
[00025795] 16
```

urbscript
Session

```
Adder.add(-2.6);
[00035411] 39.4
```

To summarize:

- Declare your object class as extending `UObject`.
- Declare a single constructor taking a `String`, and pass this string to the constructor of `UObject`.
- Declare the variables you want to share with Urbi with the type `urbi.UVar`.
- In the constructor, call `UBindVar` for each `UVar` you want as an instance variable, and `UBindFunction` for each function you want to bind.
- Don't forget to call the function `UStart` for each `UObject` class you define.

6.3 Creating new instances

When you start an Urbi server, an object of each class registered with `UStart` is created with the same name as the class. New instances can be created from Urbi using the `new` method. For each instance created in Urbi, a corresponding instance of the Java object is created. You can get the arguments passed to the constructor by defining and binding a method named `init` with the appropriate number of arguments.

For example let's add an Urbi constructor to our `Adder` class. We rewrite it as follow:

```
Java
public class Adder extends UObject // must extends UObject
{
    /// Register the class within urbi
    static { UStart(Adder.class); }

    /// Declare a variable v that will be accessible in Urbi
    private UVar v = new UVar();

    /// Constructor
    public Adder (String s)
    {
        super (s);
        UBindFunction ("init");
    }

    /// The init function is the constructor in Urbi. Here it takes
    /// one argument that we use to initialize the 'v' variable.
    /// The init function must return an int of value 0
    /// if all went OK.
    public int init (double v_init)
    {
        /// Bind the variable v to Urbi
        UBindVar (v, "v");

        /// Initialize our UVar v to the value given in the
        /// constructor
        v.setValue(v_init);

        /// Bind the function add to Urbi
        UBindFunction ("add");

        return 0;
    }

    public double add (double rhs)
    {
```

```

    /// Return the value of our UVar v (converted to double)
    /// plus the value of the argument of the function.
    return v.doubleValue () + rhs;
}
}

```

Now 'v' and 'add' are bound only when instance of the Adder object are constructed. We have added an 'init' constructor with one parameter that we use to initialize the value of v. You can run this UObject and test it in Urbi to see the difference with the previous example. Here is what it gives:

```

[00000097] *** *****
[00000097] *** Urbi SDK version 2.0 rev. 96a4b2f
[00000097] *** Copyright (C) 2004-2010 Gostai S.A.S.
[00000097] ***
[00000097] *** This program comes with ABSOLUTELY NO WARRANTY.
[00000097] *** It can be used under certain conditions.
[00000097] *** Type 'license;' or 'copyright;' for more information.
[00000097] ***
[00000097] *** Check our community site: http://www.urbiforge.org.
[00000097] ***
Adder;
[00010592] Adder
Adder.v;
[00013094:error] !!! 2.1-7: lookup failed: v
var a = Adder.new(51);
[00041405] object_13
a.v;
[00044742] 51
a.add(10);
[00054783] 61

```

urbscript
Session

6.4 Binding functions

6.4.1 Simple binding

To bind the functions to Urbi, you can use:

```
void UBindFunction (Object obj, String method_name, String[] parameters_name)
```

Java

or one of the convenient version:

```

void UBindFunction (String method_name)
void UBindFunctions(String ... method_names)
void UBindFunction (Object obj, String method_name)
void UBindFunctions (Object obj, String ... method_names)

```

Java

The first function takes as parameter the object containing the function (for now it is only possible to bind instance method, we do not handle static methods). The second parameter is the name of the function you want to bind. The third parameter is a list of the names if the types of the arguments. For example for the function add, in the previous Adder example, we could have used:

```
String[] params = { "double" };
UBindFunction (this, "add", params);
```

Java

If in your UObject you have different names for each of your methods, then you can use the shorter versions of UBindFunction.

The functions you can bind must follow these rules:

- They can have between 0 and 16 arguments.

- Their arguments can be of type:
 - Urbi types:
‘urbi.UValue’, ‘urbi.UVar’, ‘urbi.UList’, ‘urbi.UBinary’, ‘urbi.UImage’, ‘urbi.USound’, ‘urbi.UDictionary’
 - Java instances:
‘java.lang.String’, ‘java.lang.Integer’, ‘java.lang.Boolean’, ‘java.lang.Double’, ‘java.lang.Float’, ‘java.lang.Long’, ‘java.lang.Short’, ‘java.lang.Character’, ‘java.lang.Byte’
 - Java primitive types:
‘int’, ‘boolean’, ‘byte’, ‘char’, ‘short’, ‘long’, ‘float’, ‘double’.
- Their return type can be one of the following type:
 - ‘void’
 - Urbi types:
‘urbi.UValue’, ‘urbi.UVar’, ‘urbi.UList’, ‘urbi.UBinary’, ‘urbi.UImage’, ‘urbi.USound’, ‘urbi.UDictionary’
 - Java instances:
‘java.lang.String’
 - Java primitive types:
‘int’, ‘boolean’, ‘byte’, ‘char’, ‘short’, ‘long’, ‘float’, ‘double’.

6.5 Notification of a variable change or access

You can register a function that will be called each time a variable is modified by calling `UNotifyChange`, passing either an `UVar` or a variable name as first argument, and a member function of your `UObject` as second argument (and optionally a String array containing the name of the types of the parameters). The prototype for `UNotifyChange` is:

Java

```
void UNotifyChange(String var_name, String method_name, String[] args_name);
void UNotifyChange(String var_name, String method_name);
void UNotifyChange(UVar v, String method_name, String[] args_name);
void UNotifyChange(UVar v, String method_name);
```

The callback function can take zero or one argument: an `UVar` pointing to the `UVar` being modified. And the callback function must return an `int` (the value returned is currently ignored in the actual implementation) or nothing at all (`void`). The `notifyChange` callback function is always called after the variable value is changed.

Notify functions can be unregistered by calling the `unnotify` function of the `UVar` class.

6.6 Timers

The API provides two methods to have a function called periodically:

- `USetUpdate (double period)`
Sets up a timer that calls the virtual `UObject` method `update ()` with the specified `period` (in milliseconds). Disable updates if `period` is -1.
- `USetTimer (double period, Object o, String method_name)`
or
- `USetTimer (double period, Object o, String method_name, String[] args_name)`
Invoke an `UObject` member function `method_name` every `period` milliseconds.

6.7 Using Urbi variables

You can read or write any Urbi variable by creating an `UVar` passing the variable name to the constructor. Change the value by writing any compatible type to the `UVar`, and access the value by casting the `UVar` to any compatible type.

Note however that changes on the variable coming from Urbi code or an other module can take time to propagate to the `UVar`. You can read and write all the Urbi properties of an `UVar` by reading and writing the appropriate `UProp` object in the `UVar`.

6.8 Sending Urbi code

If you need to send Urbi code to the server, the `send()` function is available. You can pass it a string containing Urbi code:

```
send ("myTag:1+1;");
```

urbiscript
Session

You can also use the `call` method to make an urbiscript function call:

```
// Java equivalent of urbiscript 'System.someFunc(12, "foo");'  
call("System", "someFunc", new UValue(12), new UValue("foo"));
```

urbiscript
Session

These functions are member functions of the `UObject` class.

6.9 Providing a main class or not

We provide a main class, containing a main function, embedded in the liburbijava.jar file. This main class, called `urbi.UMain` is responsible for the loading of the liburbijava native library, and also for the registering of your uobjects.

- When launching your `UObjects` with `java` or `urbi-launch-java`, please do not provide your main class.
- When launching your `UObjects` with Eclipse:
 - You can use the `UMain` class we provide, but this class require that you pass as argument the path to your `UObject` jar file, when you run your uobject.
 - Or you can provide your own main file. In this case you have to put the `UStart(YourUObject.class)` call directly in your main file, and not in your uobject classes. Your main should look like:

```
import liburbi.main.*;  
  
public class Main  
{  
    /// load urbijava library  
    static  
    {  
        System.loadLibrary("urbijava");  
    }  
  
    public static void main(String argv[])  
    {  
        UObject.UStart(MyUObject1.class);  
        UObject.UStart(MyUObject2.class);  
        // ...  
  
        UObject.main(argv);  
    }  
}
```

Java

Call `System.loadLibrary("urbijava");` to load the liburbijava native library.

Note: when you call `System.loadLibrary`, java search for the library in the locations given in `java.library.path`. This special Java variable must be correctly set or you will get a loading error when you run your Java program. You can set this options giving: '`-Djava.library.path=path to dir containing urbijava lib`' to the Java VM running your program.

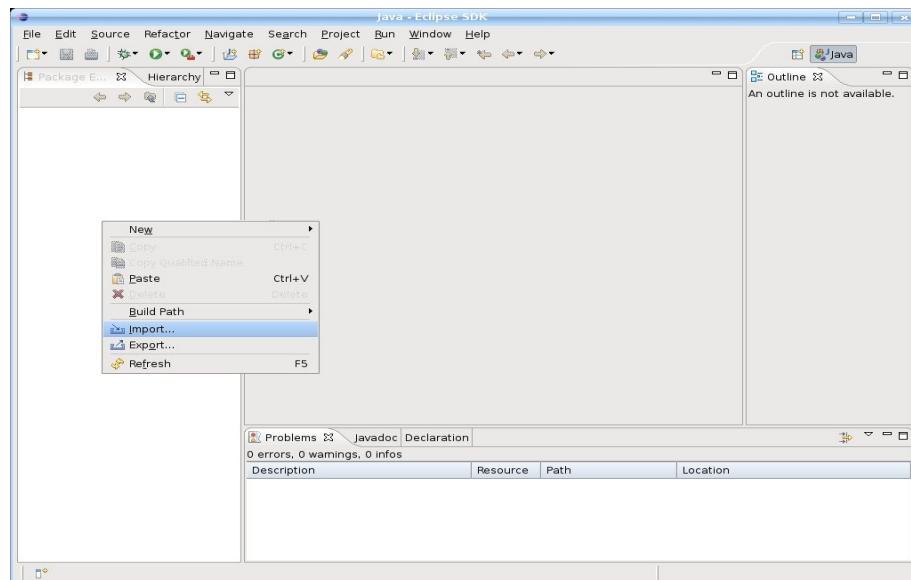
We provide two UObjects examples under eclipse. One use `urbi.UMain`, the other provide it's own main class.

6.10 Import the examples with Eclipse

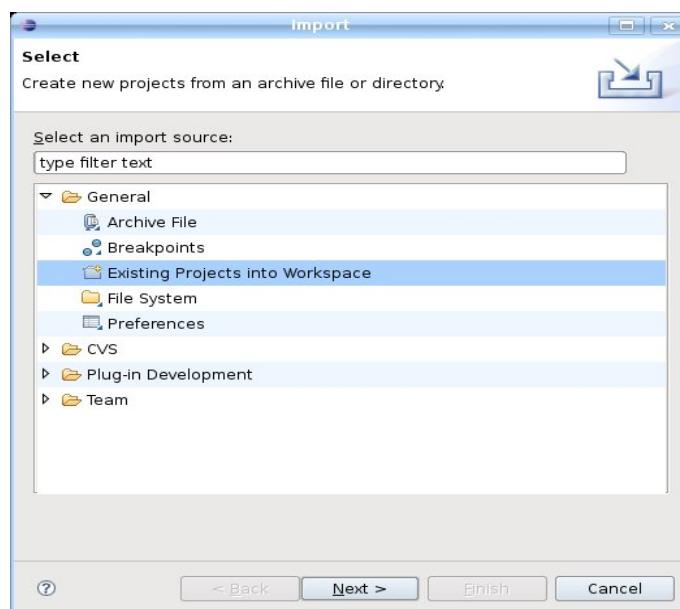
We provide a sample [Eclipse](#) project configuration that you can import in Eclipse and use to create your own UObject Java.

We illustrate here how you can do this:

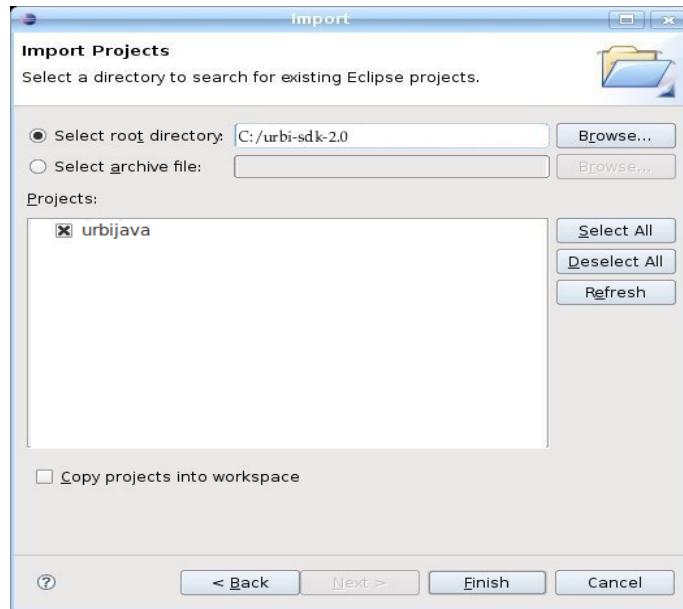
1. Open Eclipse



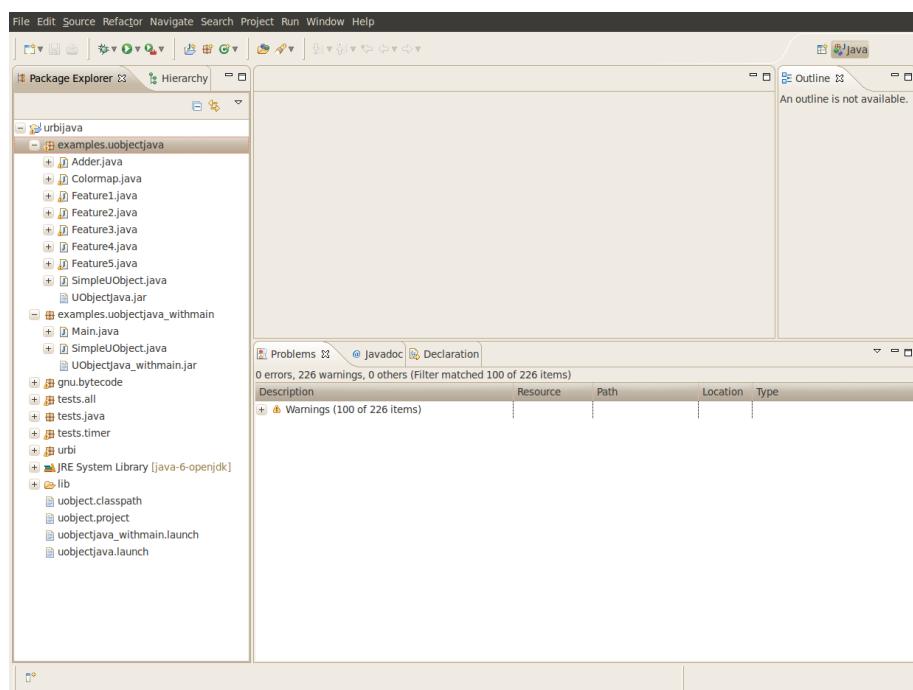
2. Right click in the Package Explorer panel and select 'import' (or go in File/import)



3. Select 'Existing Projects into Workspace' in the opened windows
4. Click 'Next'

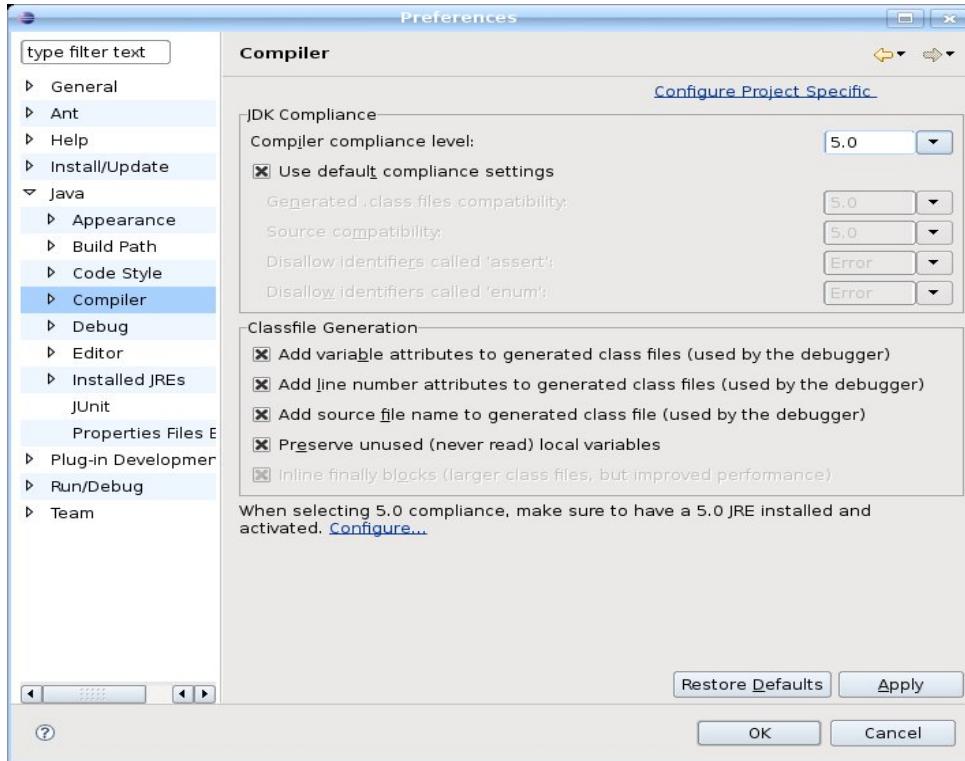


5. Enter the path of the Urbi SDK on your computer
6. Eclipse should find the .project file we provide and display the 'urbijava' project
7. Select the 'urbijava' project and click 'Finish'



The Java project is loaded. You can see the jar containing the liburbi (liburbijava.jar, storing the UObject Java API) which contains the urbi package, and also see the sources of the example we provide. We put them in the packages examples. You can inspire yourself from these examples to make your own UObjects. Here, we will see how to compile and run them in eclipse

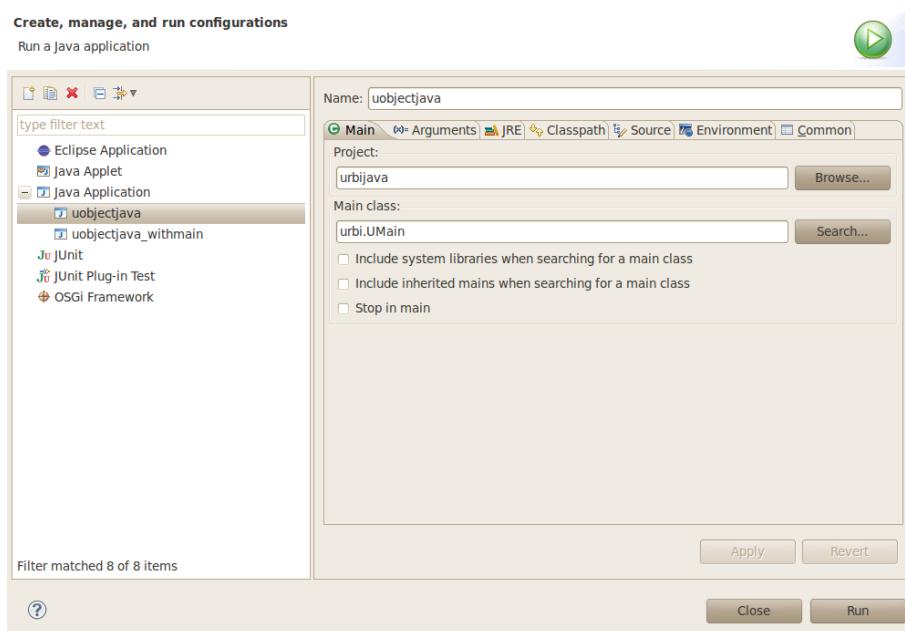
NB: If Eclipse complains about errors in the source code, it can be that your compiler compliance level is too low. You have to set the compiler compliance level to Java 5 at least (Windows/Preferences/Java/Compiler).



6.11 Run the UObject Java examples

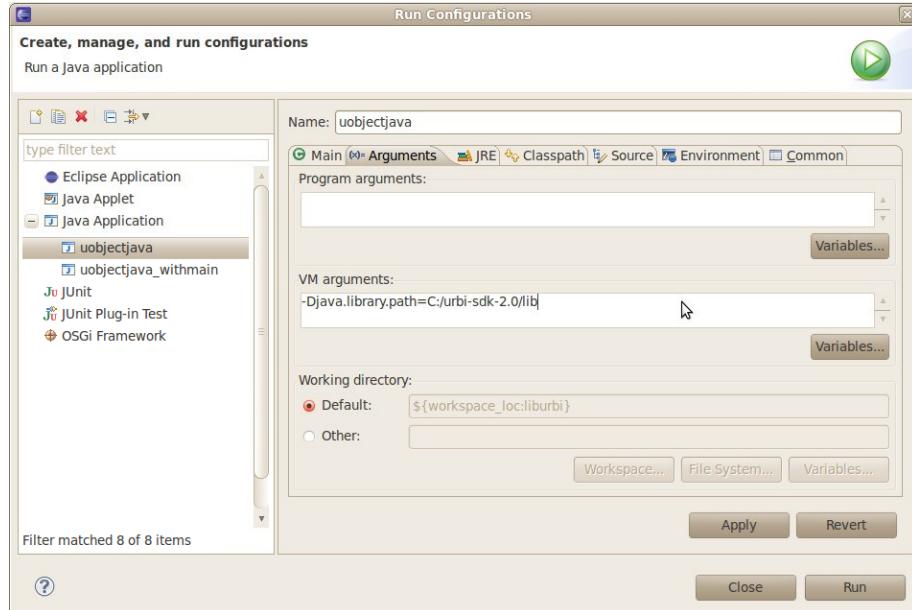
We provide a sample uobjectjava.launch files that you can load in eclipse to run the projects.

- Click on Run/Open Run Dialog (or 'Run...' or 'Run Configurations' in some versions of Eclipse)



- The launch configurations should be recognized automatically. Choose 'uobjectjava'.

- The project needs to load the urbijava native library. To this end, it will search the special java.library.path path. If this path is not correctly set, the example will trigger an error. You have to add to java.library.path the path to the *lib* folder in the Urbi SDK. You can do this from the 'Run' menu, by selecting the 'Arguments' tab, and setting -Djava.library.path=*path to lib folder* into the VM arguments. See:



- In order to run your remote UObject, you need also run an Urbi server. Your remote UObject will connect to this Urbi server. By default Urbi servers listen on port 54000, and remote UObjects try to connect to localhost on port 54000. If your urbi server is running on a different port or different address, then you will need to give these port and address as argument to your program, in the 'Arguments' tab (something like: -H address -P port). You can also write "-help" in this field, and then when you will run the program it will display some help on the arguments available.
- Click 'Apply'. Click 'Run'.

Chapter 7

Use Cases

7.1 Writing a Servomotor Device

Let's write a `UObject` for a servomotor device whose underlying API is:

- `bool initialize (int id)`
Initialize the servomotor with given ID.
- `double getPosition (int id)`
Read servomotor of given id position.
- `void setPosition (int id, double pos)`
Send a command to servomotor.
- `void setPID (int id, int p, int i, int d)`
Set P, I, and D arguments.

First our header. Our servo device provides an attribute named `val`, the standard Urbi name, and two ways to set PID gain: a method, and three variables.

```
class servo : public urbi::UObject // must inherit UObject
{
public:
    // the class must have a single constructor taking a string
    servo(const std::string&);

    // Urbi constructor
    void init(int id);

    // main attribute
    urbi::UVar val;

    // position variables:
    // P gain
    urbi::UVar P;
    // I gain
    urbi::UVar I;
    // D gain
    urbi::UVar D;

    // callback for val change
    void valueChanged(UVar& v);
    //callback for val access
    void valueAccessed(UVar& v);
    // callback for PID change
    void pidChanged(UVar& v);

    // method to change all values
```

C++

```

void setPID(int p, int i, int d);

// motor ID
int id_;
};
```

The constructor only registers init, so that our default instance `servo` does nothing, and can only be used to create new instances.

```
C++ servo::servo (const std::string& s)
: urbi::UObject (s)
{
    // register init
    UBindFunction (servo, init);
}
```

The `init` function, called in a new instance each time a new Urbi instance is created, registers the four variables (`val`, `P`, `I` and `D`), and sets up callback functions.

```
C++ // Urbi constructor.
void servo::init (int id)
{
    id_ = id;

    if (!initialize (id))
        return 1;

    UBindVar (servo, val);

    // val is both a sensor and an actuator.
    Owned (val);

    // Set blend mode to mix.
    val.blend = urbi::UMIX;

    // Register variables.
    UBindVar (servo, P);
    UBindVar (servo, I);
    UBindVar (servo, D);

    // Register functions.
    UBindFunction (servo, setPID);

    // Register callbacks on functions.
    UNotifyChange (val, &servo::valueChanged);
    UNotifyAccess (val, &servo::valueAccessed);
    UNotifyChange (P, &servo::pidChanged);
    UNotifyChange (I, &servo::pidChanged);
    UNotifyChange (D, &servo::pidChanged);
}
```

Then we define our callback methods. `servo::valueChanged` will be called each time the `val` variable is modified, just after the value is changed: we use this method to send our servo commands. `servo::valueAccessed` is called just before the value is going to be read. In this function we request the current value from the servo, and set `val` accordingly.

```
C++ // Called each time val is written to.
void servo::valueChanged (urbi::UVar& v)
{
    // v is a reference to our class member val: you can use both
    // indifferently.
    setPosition (id, (double)val);
}
```

```
// Called each time val is read.
void
servo::valueAccessed (urbi::UVar& v)
{
    // v is a reference to val.
    val = getPosition (id);
}
```

`servo::pidChanged` is called each time one of the PID variables is written to. The function `servo::setPID` can be called directly from Urbi.

```
void
servo::pidChanged (urbi::UVar& v)
{
    setPID(id, (int)P, (int)I, (int)D);
}

void
servo::setPID (int p, int i, int d)
{
    setPID (id, p, i, d);
    P = p;
    I = i;
    D = d;
}

// Register servo class to the Urbi kernel.
UStart (servo);
```

C++

That's it, compile this module, and you can use it within urbiscript:

```
// Create a new instance. Calls init (1).
headPan = new servo (1);

// Calls setPID () .
headPan.setPID (8,2,1);

// Calls valueChanged () .
headPan.val = 13;

// Calls valueAccessed () .
headPan.val * 12;

// Periodically calls valueChanged () .
headPan.val = 0 sin:1s ampli:20;

// Periodically calls valueAccessed () .
at (headPan.val < 0)
    echo ("left");
```

urbiscript
Session

The sample code above has one problem: `valueAccessed` and `valueChanged` are called each time the value is read or written from Urbi, which can happen quite often. This is a problem if sending the actual command (`setPosition` in our example) takes time to execute. There are two solutions to this issue.

7.1.1 Caching

One solution is to remember the last time the value was read/written, and not apply the new command before a fixed time. Note that the kernel is doing this automatically for `UOwned`'d variables that are in a blend mode different than `normal`. So the easiest solution to the above problem is likely to set the variable to the `mix` blending mode. The unavoidable drawback is that commands are not applied immediately, but only after a small delay.

7.1.2 Using Timers

Instead of updating/fetching the value on demand, you can chose to do it periodically based on a timer. A small difference between the two API methods comes in handy for this case: the `update()` virtual method called periodically after being set up by `USetUpdate(interval)` is called just after one pass of Urbi code execution, whereas the timers set up by `USetTimer` are called just before one pass of Urbi code execution. So the ideal solution is to read your sensors in the second callback, and write to your actuators in the first. Our previous example (omitting PID handling for clarity) can be rewritten. The header becomes:

```
C++ // Inherit from UObject.
class servo : public urbi::UObject
{
public:
    // The class must have a single constructor taking a string.
    servo (const std::string&)

    // Urbi constructor.
    void init (int id);

    // Called periodically.
    virtual int update ();
    // Called periodically.
    void getVal ();

    // Our position variable.
    urbi::UVar val;

    // Motor ID.
    int id_;
};
```

Constructor is unchanged, `init` becomes:

```
C++ // Urbi constructor.
void
servo::init (int id)
{
    id_ = id;

    if (!initialize (id))
        return 0;

    UBindVar (servo,val);
    // Val is both a sensor and an actuator.
    UOwned(val);

    // Will call update () periodically.
    USetUpdate(1);
    // Idem for getVal ().
    USetTimer (1, &servo::getVal);
}
```

`valueChanged` becomes `update` and `valueAccessed` becomes `getVal`. Instead of being called on demand, they are now called periodically. The period of the call cannot be lower than the value returned by `Object.getPeriod`; so you can set it to 0 to mean “as fast as is useful”.

7.2 Using Hubs to Group Objects

Now, suppose that, for our previous example, we can speed things up by sending all the servomotor commands at the same time, using the following method that takes two arrays of ids and positions.

```
C++
```

```
void setPositions(int count, int* ids, double* positions);
```

A hub is the perfect way to handle this task. The UObject header stays the same. We add a hub declaration:

```
class servohub : public urbi::UObjectHub
{
public:
    // The class must have a single constructor taking a string.
    servohub (const std::string&);

    // Called periodically.
    virtual int update ();

    // Called by servo.
    void addValue (int id, double val);

    int* ids;
    double* vals;
    int size;
    int count;
};
```

C++

`servo::update` becomes a call to the `addValue` method of the hub:

```
int
servo::update()
{
    ((servohub*)getUObjectHub ("servohub"))->addValue (id, (double)val);
};
```

C++

The following line can be added to the servo `init` method, although it has no use in our specific example:

```
URegister(servohub);
```

C++

Finally, the implementation of our hub methods is:

```
servohub::servohub (const std::string& s)
    : UObjectHub (s)
    , ids (0)
    , vals (0)
    , size (0)
    , count (0)
{
    // setup our timer
    USetUpdate (1);
}

int
servohub::update ()
{
    // Called periodically.
    setPositions (count, ids, vals);

    // Reset position counter.
    count = 0;

    return 0;
}

void
servohub::addValue (int id, double val)
{
    if (count + 1 < size)
```

C++

```

{
    // Allocate more memory.
    ids = (int*) realloc (ids, (count + 1) * sizeof (int));
    vals = (double*) realloc (vals, (count + 1) * sizeof (double));
    size = count + 1;
}
ids[count] = id;
vals[count++] = val;
}

UStartHub (servohub);

```

Periodically, the `update` method is called on each servo instance, which adds commands to the hub arrays, then the `update` method of the hub is called, actually sending the command and resetting the array.

7.2.1 Alternate Implementation

Alternatively, to demonstrate the use of the members `hub` variable, we can entirely remove the `update` method in the servo class (and the `USetUpdate()` call in `init`), and rewrite the hub `update` method the following way:

C++

```

int servohub::update()
{
    //called periodically
    for (UObjectList::iterator i = members.begin ();
        i != members.end ();
        ++i)
        addValue (((servo*)*i)->id, ((double)((servo*)*i)->val));
    setPositions(count, ids, vals);
    // reset position counter
    count = 0;

    return 0;
}

```

7.3 Writing a Camera Device

A camera device is an UObject whose `val` field is a binary object. The Urbi kernel itself doesn't make any difference between all the possible binary formats and data type, but the API provides image-specific structures for convenience. You must be careful about memory management. The `UBinary` structure handles its own memory: copies are deep, and the destructor frees the associated buffer. The `UIImage` and `USound` structures do not.

Let's suppose we have an underlying camera API with the following functions:

- `bool initialize (int id)`

Initialize the camera with given ID.

- `int getWidth (int id)`

Return image width.

- `int getHeight (int id)`

Return image height.

- `char* getImage (int id)`

Get image buffer of format RGB24. The buffer returned is always the same and doesn't have to be freed.

Our device code can be written as follows:

```
// Inherit from UObject.
class Camera : public urbi::UObject
{
public:
    // The class must have a single constructor taking a string.
    Camera(const std::string&);

    // Urbi constructor. Throw in case of error.
    void init (int id);

    // Our image variable and dimensions.
    urbi::UVar val;
    urbi::UVar width;
    urbi::UVar height;

    // Called on access.
    void getVal (UVar&);

    // Called periodically.
    virtual int update ();

    // Frame counter for caching.
    int frame;
    // Frame number of last access.
    int accessFrame;
    // Camera id.
    int id_;
    // Storage for last captured image.
    UBinary bin;
};
```

C++

The constructor only registers `init`:

```
Camera::Camera (const std::string& s)
    : urbi::UObject (s)
    , frame (0)
{
    UBindFunction (Camera, init);
}
```

C++

The `init` function binds the variable, a function called on access, and sets a timer up on update. It also initializes the `UBinary` structure.

```
void
Camera::init (int id)
{
    //urbi constructor
    id_ = id;
    frame = 0;
    accessFrame = 0;

    if (!initialize (id))
        throw std::runtime_error("Failed to initialize camera");

    UBindVar (Camera, val);
    UBindVar (Camera, width);
    UBindVar (Camera, height);
    width = getWidth (id);
    height = getHeight (id);

    UNotifyAccess (val, &Camera::getVal);
```

C++

```

bin.type = BINARY_IMAGE;
bin.image.width = width;
bin.image.height = height;
bin.image.imageFormat = IMAGE_RGB;
bin.image.size = width * height * 3;

// Call update () periodically.
USetUpdate (1);
}

```

The `update` function simply updates the frame counter:

C++

```

int
Camera::update ()
{
    ++frame;
    return 0;
}

```

The `getVal` updates the camera value, only if it hasn't already been called this frame, which provides a simple caching mechanism to avoid performing the potentially long operation of acquiring an image too often.

C++

```

void
Camera::getVal(urbi::UVar&)
{
    if (frame == accessFrame)
        return;

    bin.image.data = getImage (id);
    // Assign image to bin.
    val = bin;
}

UStart(Camera);

```

The image data is copied inside the kernel when proceeding this way.

Be careful, suppose that we had created the `UBinary` structure inside the `getVal` method, our buffer would have been freed at the end of the function. To avoid this, set it to 0 after assigning the `UBinary` to the `UVar`.

7.3.1 Optimization in Plugin Mode

In plugin mode, it is possible to access the buffer used by the kernel by casting the `UVar` to a `UIImage`. You can modify the content of the kernel buffer but no other argument.

7.4 Writing a Speaker or Microphone Device

Sound handling works similarly to image manipulation, the `USound` structure is provided for this purpose. The recommended way to implement a microphone is to fill the `UObject` `val` variable with the sound data corresponding to one kernel period. If you do so, the Urbi code `loop tag:micro.val`, will produce the expected result.

7.5 Writing a Softdevice: Ball Detection

Algorithms that require intense computation can be written in C++ but still be usable within Urbi: they acquire their data using `UVar` referencing other modules' variables, and output their results to other `UVar`. Let's consider the case of a ball detector device that takes an image as input, and outputs the coordinates of a ball if one is found.

The header is defined like:

```
class BallTracker : public urbi::UObject
{
public:
    BallTracker (const std::string&);
    void init (const std::string& varname);

    // Is the ball visible?
    urbi::UVar visible;

    // Ball coordinates.
    urbi::UVar x;
    urbi::UVar y;
};
```

C++

The constructor only registers `init`:

```
// The constructor registers init only.
BallTracker::BallTracker (const::string& s)
    : urbi::UObject (s)
{
    UBindFunction (BallTracker, init);
}
```

C++

The `init` function binds the variables and a callback on update of the image variable passed as a argument.

```
void
BallTracker::init (const std::string& cameraval)
{
    UBindVar (BallTracker, visible);
    UBindVar (BallTracker, x);
    UBindVar (BallTracker, y);
    UNotifyChange (cameraval, &BallTracker::newImage);

    visible = 0;
}
```

C++

The `newImage` function runs the detection algorithm on the image in its argument, and updates the variables.

```
void
BallTracker::newImage (urbi::UVar& v)
{
    // Cast to UIImage.
    urbi::UIImage i = v;
    int px,py;
    bool found = detectBall (i.data, i.width, i.height, &px, &py);

    if (found)
    {
        visible = 1;
        x = px / i.width;
        y = py / i.height;
    }
    else
        visible = 0;
}
```

C++

Part II

urbiscript User Manual

About This Part

This part, also known as the “urbiscript tutorial”, teaches the reader how to program in urbiscript. It goes from the basis to concurrent and event-based programming. No specific knowledge is expected. There is no need for a C++ compiler, as `UObject` will not be covered here (see [Part I](#)). The reference manual contains a terse and complete definition of the Urbi environment ([Part IV](#)).

Chapter 8 — First Steps

First contacts with urbiscript.

Chapter 9 — Basic Objects, Value Model

A quick introduction to objects and values.

Chapter 10 — Flow Control Constructs

Basic control flow: `if`, `for` and the like.

Chapter 11 — Advanced Functions and Scoping

Details about functions, scopes, and lexical closures.

Chapter 12 — Objective Programming, urbiscript Object Model

A more in-depth introduction to object-oriented programming in urbiscript.

Chapter 13 — Functional Programming

Functions are first-class citizens.

Chapter 14 — Parallelism, Concurrent Flow Control

The urbiscript operators for concurrency, tags.

Chapter 15 — Event-based Programming

Support for event-driven concurrency in urbiscript.

Chapter 16 — Urbi for ROS Users

How to use ROS from Urbi, and vice-versa.

Chapter 8

First Steps

This section expects that you already know how to run `urbi`. If not, please first see [Chapter 3](#).

This section introduces the most basic notions to write urbiscript code. Some aspects are presented only minimally. The goal of this section is to bootstrap yourself with the urbiscript language, to be able to study more in-depth examples afterward.

8.1 Comments

Commenting your code is crucial, so let's start by learning how to do this in urbiscript. *Comments* are ignored by the interpreter, and can be left as documentation, reminder, ... urbiscript supports C and C++ style comments:

- C style comments start with `/*` and end with `*/`. Contrary to C/C++, this type of comments does nest.
- C++ style comments start with `//` and last until the end of the line.

```
1; // This is a C++ style comment.  
[00000000] 1  
  
2 + /* This is a C-style comment. */ 2;  
[00000000] 4  
  
"foo" /* You /* can /* nest */ */ comments. */ "bar";  
[00000000] "foobar"
```

urbiscript
Session

[Chapter 3](#) introduced some of the conventions used in this document: frames such as the previous one denote “urbiscript sessions”, i.e., dialogs between Urbi and you. The output is prefixed by a number between square brackets: this is the date (in milliseconds since the server was launched) at which that line was sent by the server. This is useful at occasions, since Urbi is meant to run many parallel commands. Since these timestamps are irrelevant in documentation, they will often be filled with zeroes. More details about the typesetting of this document (and the other kinds of frames) can be found in [Chapter 35](#).

8.2 Literal values

Several special kinds of “values” can be entered directly with a specific syntax. They are called *literals*, or sometimes *manifest values*. We just met a first kind of literals: integers. There are several others, such as:

- **floats**: floating point numbers.

urbiscript
Session

```
42; // Integer literal.  
[00000000] 42  
  
3.14; // Floating point number literal.  
[00000000] 3.14
```

- **strings**: character strings.

urbiscript
Session

```
"string";  
[00000000] "string"
```

- **lists**: ordered collection of values.

urbiscript
Session

```
[1, 2, "a", "b"];  
[00000000] [1, 2, "a", "b"]
```

- **dictionaries**: unordered collection of associations.

urbiscript
Session

```
["a" => 1, "b" => 2, "3" => "three"];  
[00000000] ["3" => "three", "a" => 1, "b" => 2]
```

- **nil**: neutral value, or value placeholder. Think of it as the value that fits anywhere.

urbiscript
Session

```
nil;
```

- **void**: absence of value. Think of it as the value that fits nowhere.

urbiscript
Session

```
void;
```

These examples highlight some points:

- **Lists** and **dictionaries** in urbiscript are heterogeneous. That is, they can hold values of different types.
- The printing of **nil** and **void** is empty.
- There are many hyperlinks in this document: clicking on names such as [Dictionary](#) will drive you immediately to its specifications. This is also true for slots, such as [String.size](#). If you're looking for something, [check the index!](#)

8.3 Function calls

You can call functions with the classical, mathematical notation.

urbiscript
Session

```
cos(0); // Compute cosine  
[00000000] 1  
max(1, 3); // Get the maximum of the arguments.  
[00000000] 3  
max(1, 3, 4, 2);  
[00000000] 4
```

Again, the result of the evaluation are printed out. You can see here that function in urbiscript can be variadic, that is, take different number of arguments, such as the `max` function. Let's now try the `echo` function, that prints out its argument.

urbiscript
Session

```
echo("Hello world!");  
[00000000] *** Hello world!
```

The server prints out `Hello world!`, as expected. Note that this output is still prepended with the time stamp. Since `echo` returns `void`, no evaluation result is printed.

8.4 Variables

Variables can be introduced with the `var` keyword, given a name and an initial value. They can be assigned new values with the `=` operator.

```
var x = 42;
[00000000] 42
echo(x);
[00000000] *** 42
x = 51;
[00000000] 51
x;
[00000000] 51
```

urbiscript
Session

Note that, just as in C++, assignments return the (right-hand side) value, so you can write code like “`x = y = 0`”. The rule for valid identifiers is also the same as in C++: they may contain alphanumeric characters and underscores, but they may not start with a digit.

You may omit the initialization value, in which case it defaults to `void`.

```
var y;
y;
// Remember, the interpreter remains silent because void is printed out
// as nothing. You can convince yourself that y is actually void with
// the following methods.
y.asString;
[00000000] "void"
y.isVoid;
[00000000] true
```

urbiscript
Session

8.5 Scopes

Scopes are introduced with curly brackets (`{}`). They can contain any number of statements. Variables declared in a scope only exist within this scope.

```
{
  var x = "test";
  echo(x);
};
[00000000] *** test
// x is no longer defined here
x;
[00000073:error] !!! lookup failed: x
```

urbiscript
Session

Note that the interpreter waits for the whole scope to be input to evaluate it. Also note the mandatory terminating semicolon after the closing curly bracket.

8.6 Method calls

Methods are called on objects with the dot (`.`) notation as in C++. Method calls can be chained. Methods with no arguments don't require the parentheses.

```
0.cos();
[00000000] 1
"a-b-c".split("-");
[00000000] ["a", "b", "c"]
// Empty parentheses are optional
"foo".length();
[00000000] 3
"foo".length;
[00000000] 3
// Method call can be chained
```

urbiscript
Session

```
"".length.cos;
[00000000] 1
```

In `obj.method`, we say that `obj` is the *target*, and that we are sending him the `method message`.

8.7 Function definition

You know how to call routines, let's learn how to write some. Functions can be declared thanks to the `function` keyword, followed by the comma separated, parentheses surrounded list of formal arguments, and the body between curly brackets.

urbiscript
Session

```
// Define myFunction
function myFunction()
{
    echo("Hello world");
    echo("from my function!");
}

[00000000] function () {
    echo("Hello world");
    echo("from my function!");
}

// Invoke it
myFunction();
[00000000] *** Hello world
[00000000] *** from my function!
```

Note the strange output after you defined the function. urbiscript seems to be printing the function you just typed in again. This is because a function definition evaluates to the freshly created function.

Functions are first class citizen: they are values, just as 0 or `"foobar"`. The evaluation of a function definition yields the new function, and as always, the interpreter prints out the evaluation result, thus showing you the function again:

urbiscript
Session

```
// Work in a scope.
{
    // Define f
    function f()
    {
        echo("f")
    };
    // This does not invoke f, it returns its value.
    f;
};

[00000000] function () { echo("f") }

{
    // Define f
    function f()
    {
        echo("Hello World");
    };
    // This actually calls f
    f();
};

[00000000] *** Hello World
```

Here you can see that `f` is actually a simple value. You can just evaluate it to see its value, that is, its body. By adding the parentheses, you can actually call the function. This is a difference with methods calling, where empty parentheses are optional: method are always evaluated, you cannot retrieve their functional value — of course, you can with a different construct, but that's not the point here.

Since this output is often irrelevant, most of the time it is hidden in this documentation using the `|;` trick. When a statement is “missing”, an empty statement (`{}`) is inserted. So `code|;` is actually equivalent to `code | {};`, which means “run `code`, then run `{}` and return its value”. Since the value of `{}` is `void`, which is not displayed, this is a means to discard the result of a computation, and avoid that something is printed. Contrast the two following function definitions.

```
urbiscript
Session

function sum(a, b, c)
{
    return a + b + c;
};

[00003553] function (var a, var b, var c) { return a.'+'(b).'+'(c) }

function sum2(a, b, c)
{
    return a + b + c;
}|;
sum(20, 2, 20);
[00003556] 42
```

The `return` keyword breaks the control flow of a function (similarly to the way `break` interrupts a loop) and returns the control flow to the caller. It accepts an optional argument, the value to return to the caller.

In urbiscript, if no `return` statement is executed, the value of the last expression is returned. Actually, refrained from using `return` when you don’t need it, it is both less readable (once you get used to this programming style), and less efficient (Section 19.1.2).

```
urbiscript
Session

function succ(i) { i + 1 }|;
succ(50);
[00000000] 51
```

8.8 Conclusion

You’re now up and running with basic urbiscript code, and we can dive in details into advanced urbiscript code.

Chapter 9

Basic Objects, Value Model

In this section, we focus on urbiscript values as objects, and study urbiscript by-reference values model. We won't study classes and actual objective programming yet, these points will be presented in [Chapter 12](#).

9.1 Objects in urbiscript

An object in urbiscript is a rather simple concept: a list of slots. A *slot* is a value associated to a name. So an *object* is a list of slot names, each of which indexes a value — just like a dictionary.

```
// Create a fresh object with two slots.
class Foo
{
    var a = 42;
    var b = "foo";
};
[00000000] Foo
```

urbiscript
Session

The `localSlotNames` method lists the names of the slots of an object ([Object](#)).

```
// Inspect it.
Foo.localSlotNames;
[00000000] ["a", "asFoo", "b", "type"]
```

urbiscript
Session

You can get an object's slot value by using the dot (.) operator on this object, followed by the name of the slot.

```
// We now know the name of its slots. Let's see their value.
Foo.a;
[00000000] 42
Foo.b;
[00000000] "foo"
```

urbiscript
Session

It's as simple as this. The `inspect` method provides a convenient short-hand to discover an object ([Object](#)).

```
Foo.inspect;
[00000000] *** Inspecting Foo
[00000000] *** ** Prototypes:
[00000000] ***     Object
[00000000] *** ** Local Slots:
[00000000] ***     a : Float
[00000000] ***     asFoo : Code
[00000000] ***     b : String
[00000000] ***     type : String
```

urbiscript
Session

Let's now try to build such an object. First, we want a fresh object to work on. In urbiscript, `Object` is the parent type of every object (in fact, since urbiscript is prototype-based, `Object`

is the uppermost prototype of every object, but we'll talk about prototypes later). An instance of `Object`, is an empty, neutral object, so let's start by instantiating one with the `clone` method of `Object`.

urbiscript
Session

```
// Create the o variable as a fresh object.
var o = Object.clone;
[00000000] Object_0x00000000
// Check its content
o.inspect;
[00006725] *** Inspecting Object_0x00000000
[00006725] *** ** Prototypes:
[00006726] *** Object
[00006726] *** ** Local Slots:
```

As you can see, we obtain an empty fresh object. Note that it still inherits from `Object` features that all objects share, such as the `localSlotNames` method.

Also note how `o` is printed out: `Object_`, followed by an hexadecimal number. Since this object is empty, its printing is quite generic: its type (`Object`), and its unique identifier (every urbiscript object has one). Since these identifiers are often irrelevant and might differ between two executions, they are often filled with zeroes in this document.

We're now getting back to our empty object. We want to give it two slots, `a` and `b`, with values `42` and `"foo"` respectively. We can do this with the `setSlot` method, which takes the slot name and its value.

urbiscript
Session

```
o.setSlot("a", 42);
[00000000] 42
o.inspect;
[00009837] *** Inspecting Object_0x00000000
[00009837] *** ** Prototypes:
[00009837] *** Object
[00009838] *** ** Local Slots:
[00009838] *** a : Float
```

Here we successfully created our first slot, `a`. A good shorthand for setting slot is using the `var` keyword.

urbiscript
Session

```
// This is equivalent to o.setSlot("b", "foo").
var o.b = "foo";
[00000000] "foo"
o.inspect;
[00072678] *** Inspecting Object_0x00000000
[00072678] *** ** Prototypes:
[00072679] *** Object
[00072679] *** ** Local Slots:
[00072679] *** a : Float
[00072680] *** b : String
```

The latter form with `var` is preferred, but you need to know the name of the slot at the time of writing the code. With the former one, you can compute the slot name at execution time. Likewise, you can read a slot with a run-time determined name with the `getSlot` method, which takes the slot name as argument. The following listing illustrates the use of `getSlot` and `setSlot` to read and write slots whose names are unknown at code-writing time.

urbiscript
Session

```
function set(object, name, value)
{
    // We have to use setSlot here, since we don't
    // know the actual name of the slot.
    return object.setSlot("x_" + name, value);
};

function get(object, name)
{
    // We have to use getSlot here, since we don't
```

```
// know the actual name of the slot.
return object.getSlot("x_" + name);
};

var x = Object.clone;
[00000000] Object_0x42342448
set(x, "foo", 0);
[00000000] 0
set(x, "bar", 1);
[00000000] 1
x.localSlotNames;
[00000000] ["x_bar", "x_foo"]
get(x, "foo");
[00000000] 0
get(x, "bar");
[00000000] 1
```

Right, now we can create fresh objects, create slots in them and read them afterward, even if their name is dynamically computed, with `getSlot` and `setSlot`. Now, you might wonder if there's a method to update the value of the slot. Guess what, there's one, and it's named... `updateSlot` (originality award). Getting back to our `o` object, let's try to update one of its slots.

```
o.a;
[00000000] 42
o.updateSlot("a", 51);
[00000000] 51
o.a;
[00000000] 51
```

urbiscript
Session

Again, there's a shorthand for `updateSlot`: operator `=`.

```
o.b;
[00000000] "foo"
// Equivalent to o.updateSlot("b", "bar")
o.b = "bar";
[00000000] "bar"
o.b;
[00000000] "bar"
```

urbiscript
Session

Likewise, prefer the `'='` notation whenever possible, but you'll need `updateSlot` to update a slot whose name you don't know at code-writing time.

Note that defining the same slot twice, be it with `setSlot` or `var`, is an error. The slot must be defined once with `setSlot`, and subsequent writes must be done with `updateSlot`.

```
var o.c = 0;
[00000000] 0
// Can't redefine a slot like this
var o.c = 1;
[00000000:error] !!! slot redefinition: c
// Okay.
o.c = 1;
[00000000] 1
```

urbiscript
Session

Finally, use `removeLocalSlot` to delete a slot from an object.

```
o.localSlotNames;
[00000000] ["a", "b", "c"]
o.removeLocalSlot("c");
[00000000] Object_0x00000000
o.localSlotNames;
[00000000] ["a", "b"]
```

urbiscript
Session

Here we are, now you can inspect and modify objects at will. Don't hesitate to explore urbiscript objects you'll encounter through this documentation like this. Last point: reading, updating or removing a slot which does not exist is, of course, an error.

urbiscript
Session

```
o.d;
[00000000:error] !!! lookup failed: d
o.d = 0;
[00000000:error] !!! lookup failed: d
```

9.2 Methods

Methods in urbiscript are simply object slots containing functions. We made a little simplification earlier by saying that `obj.slot` is equivalent to `obj.getSlot("slot")`: if the fetched value is executable code such as a function, the dot form evaluates it, as illustrated below. Inside a method, `this` gives access to the target — as in C++. It can be omitted if there is no ambiguity with local variables.

urbiscript
Session

```
var o = Object.clone;
[00000000] Object_0x0
// This syntax stores the function in the 'f' slot of 'o'.
function o.f ()
{
    echo("This is f with target " + this);
    return 42;
}|;
// The slot value is the function.
o.getSlot("f");
[00000001] function () {
    echo("This is f with target ".+'(this)');
    return 42;
}
// Huho, the function is invoked!
o.f;
[00000000] *** This is f with target Object_0x0
[00000000] 42
// The parentheses are in fact optional.
o.f();
[00000000] *** This is f with target Object_0x0
[00000000] 42
```

This was designed this way so as one can replace an attribute, such as an integer, with a function that computes the value. This enables to replace an attribute with a method without changing the object interface, since the parentheses are optional.

This implies that `getSlot` can be a better tool for object inspection to avoid invoking slots, as shown below.

urbiscript
Session

```
// The 'empty' method of strings returns whether the string is empty.
"foo".empty;
[00000000] false
"".empty;
[00000000] true
// Using getSlot, we can fetch the function without calling it.
"".getSlot("asList");
[00000000] function () { split("") }
```

The `asList` function simply bounces the task to `split`. Let's try `getSlot`'ing another method:

urbiscript
Session

```
"foo".size;
[00000000] 3
"foo".getSlot("size");
[00000000] Primitive_0x422f0908
```

The `size` method of `String` is another type of object: a `Primitive`. These objects are executable, like functions, but they are actually opaque primitives implemented in C++.

9.3 Everything is an object

If you're wondering what is an object and what is not, the answer is simple: every single bit of value you manipulate in urbiscript is an object, including primitive types, types themselves, functions, ...

```
var x = 0;
[00000000] 0
x.localSlotNames;
[00000000] []
var x.slot = 1;
[00000000] 1
x.localSlotNames;
[00000000] ["slot"]
x.slot;
[00000000] 1
x;
[00000000] 0
```

urbiscript
Session

As you can see, integers are objects just like any other value.

9.4 The urbiscript values model

We are now going to focus on the urbiscript value model, that is how values are stored and passed around. The whole point is to understand when variables point to the same object. For this, we introduce `uid`, a method that returns the target's unique identifier — the same one that was printed when we evaluated `Object.clone`. Since uids might vary from an execution to another, their values in this documentation are dummy, yet not null to be able to differentiate them.

```
var o = Object.clone;
[00000000] Object_0x100000
o.uid;
[00000000] "0x100000"
42.uid;
[00000000] "0x200000"
42.uid;
[00000000] "0x300000"
```

urbiscript
Session

Our objects have different uids, reflecting the fact that they are different objects. Note that entering the same integer twice (42 here) yields different objects. Let's introduce new operators before diving in this concept. First the equality operator: `==`. This operator is the exact same as C or C++'s one, it simply returns whether its two operands are *semantically* equal. The second operator is `====`, which is the *physical* equality operator. It returns whether its two operands are the same object, which is equivalent to having the same uid. This can seem a bit confusing; let's have an example.

```
var a = 42;
[00000000] 42
var b = 42;
[00000000] 42
a == b;
[00000000] true
a === b;
[00000000] false
```

urbiscript
Session

Here, the `==` operator reports that `a` and `b` are equal — indeed, they both evaluate to 42. Yet, the `==` operator shows that they are not the same object: they are two different instances of integer objects, both equal 42.

Thanks to this operator, we can point out the fact that slots and local variables in urbiscript have a reference semantic. That is, when you defining a local variable or a slot, you're not copying any value (as you would be in C or C++), you're only making it refer to an already existing value (as you would in Ruby or Java).

urbiscript
Session

```
var a = 42;
[00000000] 42
var b = 42;
[00000000] 42
var c = a; // c refers to the same object as a.
[00000000] 42
// a, b and c are equal: they have the same value.
a == b && a == c;
[00000000] true
// Yet only a and c are actually the same object.
a === b;
[00000000] false
a === c;
[00000000] true
```

So here we see that `a` and `c` point to the same integer, while `b` points to a second one. This a non-trivial fact: any modification on `a` will affect `c` as well, as shown below.

urbiscript
Session

```
a.localSlotNames;
[00000000] []
b.localSlotNames;
[00000000] []
c.localSlotNames;
[00000000] []
var a.flag; // Create a slot in a.
a.localSlotNames;
[00000000] ["flag"]
b.localSlotNames;
[00000000] []
c.localSlotNames;
[00000000] ["flag"]
```

Updating slots or local variables does not update the referenced value. It simply redirects the variable to the new given value.

urbiscript
Session

```
var a = 42;
[00000000] 42
var b = a;
[00000000] 42
// b and a point to the same integer.
a === b;
[00000000] true
// Updating b won't change the referred value, 42,
// it makes it reference a fresh integer with value 51.
b = 51;
[00000000] 51
// Thus, a is left unchanged:
a;
[00000000] 42
```

Understanding the two latter examples is really important, to be aware of what your variable are referring to.

Finally, function and method arguments are also passed by reference: they can be modified by the function.

urbiscript
Session

```
function test(arg)
```

```
{
  var arg.flag; // add a slot in arg
  echo(arg.uid); // print its uid
}|;
var x = Object.clone;
[00000000] Object_0x1
x.uid;
[00000000] "0x1"
test(x);
[00000000] *** 0x1
x.localSlotNames;
[00000000] ["flag"]
```

Beware however that arguments are passed by reference, and the behavior might not be what you may expected.

```
function test(arg)
{
  // Updates the local variable arg to refer 1.
  // Does not affect the referred value, nor the actual external argument.
  arg = 1;
}|;
var x = 0;
[00000000] 0
test(x);
[00000000] 1
// x wasn't modified
x;
[00000000] 0
```

urbiscript
Session

9.5 Conclusion

You should now understand the reference semantic of local variables, slots and arguments. It's very important to keep them in mind, otherwise you will end up modifying variables you didn't want, or change a copy of reference, failing to update the desired one.

Chapter 10

Flow Control Constructs

In this section, we'll introduce some flow control structures that will prove handy later. Most of them are inspired by C/C++.

10.1 if

The `if` construct is the same has C/C++'s one. The `if` keyword is followed by a condition between parentheses and an expression, and optionally the `else` keyword and another expression. If the condition evaluates to true, the first expression is evaluated. Otherwise, the second expression is evaluated if present.

```
if (true)
    echo("ok");
[00000000] *** ok
if (false)
    echo("ko")
else
    echo("ok");
[00000000] *** ok
```

urbiscript
Session

The `if` construct is an expression: it has a value.

```
echo({ if (false) "a" else "b" });
[00000000] *** b
```

urbiscript
Session

10.2 while

The `while` construct is, again, the same as in C/C++. The `while` keyword is followed by a condition between parentheses and an expression. If the condition evaluation is false, the execution jumps after the while block; otherwise, the expression is evaluated and control jumps before the while block.

```
var x = 2;
[00000000] 2
while (x < 40)
{
    x += 10;
    echo(x);
}
[00000000] *** 12
[00000000] *** 22
[00000000] *** 32
[00000000] *** 42
```

urbiscript
Session

10.3 for

The `for` keyword supports different constructs, as in languages such as Java, C#, or even the forthcoming C++ revision.

The first construct is hardly more than syntactic sugar for a `while` loop.

urbiscript
Session

```
for (var x = 2; x < 40; x += 10)
    echo(x);
[00000000] *** 2
[00000000] *** 12
[00000000] *** 22
[00000000] *** 32
```

The second construct allows to iterate over members of a collection, such as a list. The `for` keyword, followed by `var`, an identifier, a colon (or `in`), an expression and a scope, executes the scope for every element in the collection resulting of the evaluation of the expression, with the variable named with the identifier referring to the list members.

urbiscript
Session

```
for (var e : [1, 2, 3]) { echo(e) };
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
```

10.4 switch

The syntax of the `switch` construct is similar to C/C++'s one, except it works on any kind of object, not only integral ones. Comparison is done by semantic equality (operator `==`). Execution will jump out of the `switch`-block after a case has been executed (no need to `break`). Also, contrary to C++, the whole construct has a value: that of the matching `case`.

urbiscript
Session

```
switch ("bar")
{
    case "foo": 0;
    case "bar": 1;
    case "baz": 2;
    case "qux": 3;
};
[00000000] 1
```

10.5 do

A `do` scope is a shorthand to perform several actions on an object.

urbiscript
Session

```
var o1 = Object.clone;
[00000000] Object_0x423a0708
var o1.one = 1;
[00000000] 1
var o1.two = 2;
[00000000] 2
echo(o1.uid);
[00000000] *** 0x423a0708
```

The same result can be obtained with a short `do` scope, that redirect method calls to their target, as in the listing below. This is similar to the Pascal “`with`” construct. The value of the `do`-block is the target itself.

urbiscript
Session

```
var o2 = Object.clone;
[00000000] Object_0x42339e08
// All the message in this scope are destined to o.
do (o2)
```

```
{  
    var one = 1; // var is a shortcut for the setSlot  
    var two = 2; // message, so it applies on obj too.  
    echo(uid);  
};  
[00000000] *** 0x42339e08  
[00000000] Object_0x42339e08
```


Chapter 11

Advanced Functions and Scoping

This section presents advanced uses of functions and scoping, as well as their combo: lexical closures, which prove to be a very powerful tool.

11.1 Scopes as expressions

Contrary to other languages from the C family, scopes are expressions: they can be used where values are expected, just as `1 + 1` or `"foo"`. They evaluate to the value of their last expression, or `void` if they are empty. The following listing illustrates the use of scopes as expressions. The last semicolon inside a scope is optional.

```
// Scopes evaluate to the value of their last expression.  
{ 1; 2; 3; };  
[00000000] 3  
// They are expressions.  
echo({1; 2; 3});  
[00000000] *** 3
```

urbiscript
Session

11.2 Advanced scoping

Scopes can be nested. Variables can be redefined in nested scopes. In this case, the inner variables hide the outer ones, as illustrated below.

```
var x = 0; // Define the outer x.  
[00000000] 0  
{  
    var x = 1; // Define an inner x.  
    x = 2; // These refer to  
    echo(x); // the inner x  
};  
[00000000] *** 2  
x; // This is the outer x again.  
[00000000] 0  
{  
    x = 3; // This is still the outer x.  
    echo(x);  
};  
[00000000] *** 3  
x;  
[00000000] 3
```

urbiscript
Session

11.3 Local functions

Functions can be defined anywhere local variables can — that is, about anywhere. These functions' visibility are limited to the scope they're defined in, like variables. This enables for instance to write local helper functions like `max2` in the example below.

```
urbiscript
Session
function max3(a, b, c) // Max of three values
{
    function max2(a, b)
    {
        if (a > b)
            a
        else
            b
    };
    max2(a, max2(b, c));
}!;
```

11.4 Lexical closures

A *closure* is the capture by a function of a variable external to this function. urbiscript supports lexical closure: functions can refer to outer local variables, as long as they are visible (in scope) from where the function is defined.

```
urbiscript
Session
function printSalaries(var rate)
{
    var charges = 100;
    function computeSalary(var hours)
    {
        // rate and charges are captured from the environment by closure.
        rate * hours - charges
    };

    echo("Alice's salary is " + computeSalary(35));
    echo("Bob's salary is " + computeSalary(30));
}!;
printSalaries(15);
[00000000] *** Alice's salary is 425
[00000000] *** Bob's salary is 350
```

Closures can also change captured variables, as shown below.

```
urbiscript
Session
var a = 0;
[00000000] 0
var b = 0;
[00000000] 0
function add(n)
{
    // a and b are updated by closure.
    a += n;
    b += n;
    {}
}!;
add(25);
add(25);
add(1);
a;
[00000000] 51
b;
[00000000] 51
```

Closure can be really powerful tools in some situations; they are even more useful when combined with functional programming, as described in [Chapter 13](#).

Chapter 12

Objective Programming, urbiscript Object Model

This section presents object programming in urbiscript: the prototype-based object model of urbiscript, and how to define and use classes.

12.1 Prototype-Based Programming in urbiscript

You're probably already familiar with class-based object programming, since this is the C++, Java, C# model. Classes and objects are very different entities. Classes and types are static entities that do not exist at run-time, while objects are dynamic entities that do not exist at compile time.

Prototype-based object programming is different: the difference between classes and objects, between types and values, is blurred. Instead, you have an object, that is already an instance, and that you might clone to obtain a new one that you can modify afterward. Prototype-based programming was introduced by the Self language, and is used in several popular script languages such as Io or JavaScript.

Class-based programming can be considered with an industrial metaphor: classes are molds, from which objects are generated. Prototype-based programming is more biological: a prototype object is cloned into another object which can be modified during its lifetime.

Consider pairs for instance (see [Pair](#)). Pairs hold two values, `first` and `second`, like an `std::pair` in C++. Since urbiscript is prototype-based, there is no pair class. Instead, `Pair` is really a pair (object).

```
Pair;
[00000000] (nil, nil)
```

urbiscript
Session

We can see here that `Pair` is a pair whose two values are equal to `nil` — which is a reasonable default value. To get a pair of our own, we simply clone `Pair`. We can then use it as a regular pair.

```
var p = Pair.clone;
[00000000] (nil, nil)
p.first = "101010";
[00000000] "101010"
p.second = true;
[00000000] true
p;
[00000000] ("101010", true)
Pair;
[00000000] (nil, nil)
```

urbiscript
Session

Since `Pair` is a regular pair object, you can modify and use it at will. Yet this is not a good idea, since you will alter your base prototype, which alters any derivative, future and even past.

urbiscript
Session

```
var before = Pair.clone;
[00000000] (nil, nil)
Pair.first = false;
[00000000] false
var after = Pair.clone;
[00000000] (false, nil)
before;
[00000000] (false, nil)
// before and after share the same first: that of Pair.
assert(Pair.first === before.first);
assert(Pair.first === after.first);
```

12.2 Prototypes and Slot Lookup

In prototype-based language, *is-a* relations (being an instance of some type) and inheritance relations (extending another type) are simplified in a single relation: prototyping. You can inspect an object prototypes with the `protos` method.

urbiscript Session

```
var p = Pair.clone;
[00000000] (nil, nil)
p.protos;
[00000000] [(nil, nil)]
```

As expected, our fresh pair has one prototype, `(nil, nil)`, which is how `Pair` displays itself. We can check this as presented below.

urbiscript Session

```
// List.head returns the first element.
p.protos.head;
[00000000] (nil, nil)
// Check that the prototype is really Pair.
p.protos.head === Pair;
[00000000] true
```

Prototypes are the base of the slot lookup mechanism. Slot lookup is the action of finding an object slot when the dot notation is used. So far, when we typed `obj.slot`, `slot` was always a slot of `obj`. Yet, this call can be valid even if `obj` has no `slot` slot, because slots are also looked up in prototypes. For instance, `p`, our clone of `Pair`, has no `first` or `second` slots. Yet, `p.first` and `p.second` work, because these slots are present in `Pair`, which is `p`'s prototype. This is illustrated below.

urbiscript Session

```
var p = Pair.clone;
[00000000] (nil, nil)
// p has no slots of its own.
p.localSlotNames;
[00000000] []
// Yet this works.
p.first;
// This is because p has Pair for prototype, and Pair has a 'first' slot.
p.protos.head === Pair;
[00000000] true
"first" in Pair.localSlotNames && "second" in Pair.localSlotNames;
[00000000] true
```

As shown here, the `clone` method simply creates an empty object, with its target as prototype. The new object has the exact same behavior as the cloned one thanks to slot lookup.

Let's experience slot lookup by ourselves. In urbiscript, you can add and remove prototypes from an object thanks to `addProto` and `removeProto`.

urbiscript Session

```
// We create a fresh object.
var c = Object.clone;
[00000000] Object_0x1
```

```
// As expected, it has no 'slot' slot.
c.slot;
[00000000:error] !!! lookup failed: slot
var p = Object.clone;
[00000000] Object_0x2
var p.slot = 0;
[00000000] 0
c.addProto(p);
[00000000] Object_0x1
// Now, 'slot' is found in c, because it is inherited from p.
c.slot;
[00000000] 0
c.removeProto(p);
[00000000] Object_0x1
// Back to our good old lookup error.
c.slot;
[00000000:error] !!! lookup failed: slot
```

The slot lookup algorithm in urbiscript in a depth-first traversal of the object prototypes tree. Formally, when the *s* slot is requested from *x*:

- If *x* itself has the slot, the requested value is found.
- Otherwise, the same lookup algorithm is applied on all prototypes, most recent first.

Thus, slots from the last prototype added take precedence over other prototype's slots.

```
var proto1 = Object.clone;
[00000000] Object_0x10000000
var proto2 = Object.clone;
[00000000] Object_0x20000000
var o = Object.clone;
[00000000] Object_0x30000000
o.addProto(proto1);
[00000000] Object_0x30000000
o.addProto(proto2);
[00000000] Object_0x30000000
// We give o an x slot through proto1.
var proto1.x = 0;
[00000000] 0
o.x;
[00000000] 0
// proto2 is visited first during lookup.
// Thus its "x" slot takes precedence over proto1's.
var proto2.x = 1;
[00000000] 1
o.x;
[00000000] 1
// Of course, o's own slots have the highest precedence.
var o.x = 2;
[00000000] 2
o.x;
[00000000] 2
```

urbiscript
Session

You can check where in the prototype hierarchy a slot is found with the `locateSlot` method. This is a very handful tool when inspecting an object.

```
var p = Pair.clone;
[00000000] (nil, nil)
// Check that the 'first' slot is found in Pair
p.locateSlot("first") === Pair;
[00000000] true
// Where does locateSlot itself come from? Object itself!
p.locateSlot("locateSlot");
[00000000] Object
```

urbiscript
Session

The prototype model is rather simple: creating a fresh object simply consists in cloning a model object, a prototype, that was provided to you. Moreover, you can add behavior to an object at any time with a simple `addProto`: you can make any object a fully functional `Pair` with a simple `myObj.addProto(Pair)`.

12.3 Copy on Write

One point might be bothering you though: what if you want to update a slot value in a clone of your prototype?

Say we implement a simple prototype, with an `x` slot equal to 0, and clone it twice. We have three objects with an `x` slot, yet only one actual 0 integer. Will modifying `x` in one of the clone change the prototype's `x`, thus altering the prototype and the other clone as well?

The answer is, of course, no, as illustrated below.

urbiscript
Session

```
var proto = Object.clone;
[00000000] Object_0x1
var proto.x = 0;
[00000000] 0
var o1 = proto.clone;
[00000000] Object_0x2
var o2 = proto.clone;
[00000000] Object_0x3
// Are we modifying proto's x slot here?
o1.x = 1;
[00000000] 1
// Obviously not
o2.x;
[00000000] 0
proto.x;
[00000000] 0
o1.x;
[00000000] 1
```

This work thanks to *copy-on-write*: slots are first duplicated to the local object when they're updated, as we can check below.

urbiscript
Session

```
// This is the continuation of previous example.

// As expected, o2 finds "x" in proto
o2.locateSlot("x") === proto;
[00000000] true
// Yet o1 doesn't anymore
o1.locateSlot("x") === proto;
[00000000] false
// Because the slot was duplicated locally
o1.locateSlot("x") === o1;
[00000000] true
```

This is why, when we cloned `Pair` earlier, and modified the “first” slot of our fresh `Pair`, we didn't alter `Pair` one all its other clones.

12.4 Defining Pseudo-Classes

Now that we know the internals of urbiscript's object model, we can start defining our own classes.

But wait, we just said there are no classes in prototype-based object-oriented languages! That is true: there are no classes in the sense of C++, i.e., compile-time entities that are not objects. Instead, prototype-based languages rely on the existence of a canonical object (the *prototype*) from which (pseudo) *instances* are derived. Yet, since the syntactic inspiration for

urbiscript comes from languages such as Java, C++ and so forth, it is nevertheless the `class` keyword that is used to define the pseudo-classes, i.e., prototypes.

As an example, we define our own `Pair` class. We just have to create a pair, with its `first` and `second` slots. For this we use the `do` scope described in [Section 10.5](#). The listing below defines a new `Pair` class. The `asString` function is simply used to customize pairs printing — don't give it too much attention for now.

```
var MyPair = Object.clone;
[00000000] Object_0x1
do (MyPair)
{
  var first = nil;
  var second = nil;
  function asString ()
  {
    "MyPair: " + first + ", " + second
  };
}|;
// We just defined a pair
MyPair;
[00000000] MyPair: nil, nil
// Let's try it out
var p = MyPair.clone;
[00000000] MyPair: nil, nil
p.first = 0;
[00000000] 0
p;
[00000000] MyPair: 0, nil
MyPair;
[00000000] MyPair: nil, nil
```

urbiscript
Session

That's it, we defined a pair that can be cloned at will! urbiscript provides a shorthand to define classes as we did above: the `class` keyword.

```
class MyPair
{
  var first = nil;
  var second = nil;
  function asString() { "(" + first + ", " + second + ")"; }
};
[00000000] (nil, nil)
```

urbiscript
Session

The `class` keyword simply creates `MyPair` with `Object.clone`, and provides you with a `do (MyPair)` scope. It actually also pre-defines a few slots, but this is not the point here.

It is also possible to specify a proto for the newly created “class”, using the same syntax as Java and C++:

```
class Top
{
  var top = "top";
};
[00000000] Top

class Bottom : Top
{
  var bottom = "bottom";
};
[00000000] Bottom

Bottom.new.top;
[00000000] "top"
```

urbiscript
Session

For more details, see [Section 23.1.6.8](#).

12.5 Constructors

As we've seen, we can use the `clone` method on any object to obtain an identical object. Yet, some classes provide more elaborate constructors, accessible by calling `new` instead of `clone`, potentially passing arguments.

urbiscript Session

```
var p = Pair.new("foo", false);
[00000000] ("foo", false)
```

While `clone` guarantees you obtain an empty fresh object inheriting from the prototype, `new` behavior is left to the discretion of the cloned prototype — although its behavior is the same as `clone` by default.

To define such constructors, prototypes only need to provide an `init` method, that will be called with the arguments given to `new`. For instance, we can improve our previous `Pair` class with a constructor.

urbiscript Session

```
class MyPair
{
    var first = nil;
    var second = nil;
    function init(f, s) { first = f; second = s; };
    function asString() { "(" + first + ", " + second + ")"; };
};

[00000000] (nil, nil)
MyPair.new(0, 1);
[00000000] (0, 1)
```

12.6 Operators

In urbiscript, operators such as `+`, `&&` and others, are regular functions that benefit from a bit of syntactic sugar. To be more precise, `a+b` is exactly the same as `a.'+'(b)`. The rules to resolve slot names apply too, i.e., the `'+'` slot is looked for in `a`, then in its prototypes.

The following example provides arithmetic between pairs.

urbiscript Session

```
class ArithPair
{
    var first = nil;
    var second = nil;
    function init(f, s) { first = f; second = s; };
    function asString() { "(" + first + ", " + second + ")"; };
    function '+'(rhs) { new(first + rhs.first, second + rhs.second); };
    function '-'(rhs) { new(first - rhs.first, second - rhs.second); };
    function '*'(rhs) { new(first * rhs.first, second * rhs.second); };
    function '/'(rhs) { new(first / rhs.first, second / rhs.second); };
};

[00000000] (nil, nil)
ArithPair.new(1, 10) + ArithPair.new(2, 20) * ArithPair.new(3, 30);
[00000000] (7, 610)
```

12.7 Properties

Slots should be understood as names to values. Several names may point to the same value (a phenomenon called *aliasing*). Enriching values is easy: provide them with new slots. Yet it is sometimes needed to define the behavior of the *slot* rather than the property of the value. This is the purpose of *properties* in urbiscript.

12.7.1 Features of Values

In following example, we attach some random property `foo` to the value pointed to by the slot `x`.

```
var x = 123;
[00000000] 123
var x.foo = 42;
[00000000] 42
```

urbiscript
Session

If `y` is another slot to the value of `x`, then it provides the same `foo` feature:

```
var y = x;
[00000000] 123
y.foo;
[00000000] 42
```

urbiscript
Session

If `x` is bound to a new object (e.g., 456), then the feature `foo` is no longer present, since it's a feature of the *value* (i.e., 123), and not one of the slot (i.e., `x`).

```
x = 456;
[00000000] 456
x.foo;
[00000000:error] !!! lookup failed: foo
```

urbiscript
Session

Of course, `y`, which is still linked to the original value (123), answers to queries to `foo`.

```
y.foo;
[00000000] 42
```

urbiscript
Session

12.7.2 Features of Slots

If, on the contrary you want to attach a feature to the slot-as-a-name, rather than to the value it contains, use the *properties*. The syntax is `slotName->propertyName`.

```
x = 123;
[00000000] 123
x->foo = 42;
[00000000] 42
x->foo;
[00000000] 42
```

urbiscript
Session

Copying the value contained by a slot does *not* propagate the properties of the slot:

```
y = x;
[00000000] 123
y->foo;
[00000000:error] !!! property lookup failed: y->foo
```

urbiscript
Session

And if you assign a new value to a slot, the properties of the slot are preserved:

```
x = 456;
[00000000] 456
x->foo = 42;
[00000000] 42
```

urbiscript
Session

Chapter 13

Functional Programming

urbiscript support functional programming through first class functions and lambda expressions.

13.1 First class functions

urbiscript has first class functions, i.e., functions are regular values, just like integers or strings. They can be stored in variables, passed as arguments to other functions, and so forth. For instance, you don't need to write `function object.f()/* ... */` to insert a function in an object, you can simply use `setSlot`.

```
var o = Object.clone();
// Here we can use f as any regular value.
o.setSlot("m1", function () { echo("Hello") })|;
// This is strictly equivalent.
var o.m2 = function () { echo("Hello") }|;
o.m1;
[00000000] *** Hello
o.m2;
[00000000] *** Hello
```

urbiscript
Session

This enables to write powerful pieces of code, like functions that take function as argument. For instance, consider the `all` function: given a list and a function, it applies the function to each element of the list, and returns whether all calls returned true. This enables to check very simply if all elements in a list verify a predicate.

```
function all(list, predicate)
{
    for (var elt : list)
        if (!predicate(elt))
            return false;
        return true;
}|;
// Check if all elements in a list are positive.
function positive(x) { x >= 0 }|;
all([1, 2, 3], getSlot("positive"));
[00000000] true
all([1, 2, -3], getSlot("positive"));
[00000000] false
```

urbiscript
Session

It turns out that `all` already exists: instead of `all(list, predicate)`, use `list.all(predicate)`, see [RangeIterable.all](#).

13.2 Lambda functions

Another nice feature is the ability to write lambda functions, which are anonymous functions. You can create a functional value as an expression, without naming it, with the syntax shown

below.

urbiscript
Session

```
// Create an anonymous function
function (x) {x + 1}|;
// This enable to easily pass function
// to our "all" function:
[1, 2, 3].all(function (x) { x > 0});
[00000000] true
```

In fact, the `function` construct we saw earlier is only a shorthand for a variable assignment.

urbiscript
Session

```
// This ...
function obj.f /*...*/ {/*...*/};
// ... is actually a shorthand for:
const var obj.f = function /*...*/ /* ... */;
```

13.3 Lazy arguments

Most popular programming languages use strict arguments evaluation: arguments are evaluated before functions are called. Other languages use lazy evaluation: argument are evaluated by the function only when needed. In urbiscript, evaluation is strict by default, but you can ask a function not to evaluate its arguments, and do it by hand. This works by not specifying formal arguments. The function is provided with a `call` object that enables you to evaluate arguments.

urbiscript
Session

```
// Note the lack of formal arguments specification
function first
{
    // Evaluate only the first argument.
    call.evalArgAt(0);
};

first(echo("first"), echo("second"));
[00000000] *** first
function reverse
{
    call.evalArgAt(1);
    call.evalArgAt(0);
};

reverse(echo("first"), echo("second"));
[00000000] *** second
[00000000] *** first
```

A good example are logic operators. Although C++ is a strict language, it uses a few logic operators. For instance, the logical and (`&&`) does not evaluate its right operand if the left operand is false (the result will be false anyway).

urbiscript logic operator mimic this behavior. The listing below shows how one can implement such a behavior.

urbiscript
Session

```
function myAnd
{
    if (call.evalArgAt(0))
        call.evalArgAt(1)
    else
        false;
};

function f()
{
    echo("f executed");
    return true;
};

myAnd(false, f());
```

```
[00000000] false
myAnd(true, f());
[00000000] *** f executed
[00000000] true
```


Chapter 14

Parallelism, Concurrent Flow Control

Parallelism is a major feature of urbiscript. So far, all we've seen already existed in other languages — although we tried to pick, mix and adapt features and paradigms to create a nice scripting language. Parallelism is one of the corner stones of its paradigm, and what makes it so well suited to high-level scripting of interactive agents, in fields such as robotics or AI.

14.1 Parallelism operators

For now, we've separated our different commands with a semicolon (;). There are actually four statement separators in urbiscript:

- “;”: Serialization operator. Wait for the left operand to finish before continuing.
- “&”: Parallelism n-ary operator. All its operands are started simultaneously, and executed in parallel. The & block itself finishes when all the operands have finished. & has higher precedence than other separators.
- “,”: Background operator. Its left operand is started, and then it proceeds immediately to its right operand. This operator is bound to scopes: when used inside a scope, the scope itself finishes only when all the statements backgrounded with ‘,’ have finished.

The example below demonstrates the use of & to launch two functions in parallel.

```
function test(name)
{
    echo(name + ": 1");
    echo(name + ": 2");
    echo(name + ": 3");
} |;
// Serialized executions
test("left") ; test ("middle"); test ("right");
[00000000] *** left: 1
[00000000] *** left: 2
[00000000] *** left: 3
[00000000] *** middle: 1
[00000000] *** middle: 2
[00000000] *** middle: 3
[00000000] *** right: 1
[00000000] *** right: 2
[00000000] *** right: 3
// Parallel execution
test("left") & test("middle") & test ("right");
[00000000] *** left: 1
[00000000] *** middle: 1
```

urbiscript
Session

```
[00000000] *** right: 1
[00000000] *** left: 2
[00000000] *** middle: 2
[00000000] *** right: 2
[00000000] *** left: 3
[00000000] *** middle: 3
[00000000] *** right: 3
```

In this test, we see that the `&` runs its operands simultaneously.

The difference between “`&`” and “`,`” is rather subtle:

- In the top level, no operand of a job will start “`&`” until all are known. So if you send a line ending with “`&`”, the system will wait for the right operand (in fact, it will wait for a “`,`” or a “`;`”) before firing its left operand. A statement ending with “`,`” will be fired immediately.
- Execution is blocked after a “`&`” group until all its children have finished.

urbiscript
Session

```
function test(name)
{
    echo(name + ": 1");
    echo(name + ": 2");
    echo(name + ": 3");
}
// Run test and echo("right") in parallel,
// and wait until both are done before continuing
test("left") & echo("right"); echo("done");
[00000000] *** left: 1
[00000000] *** right
[00000000] *** left: 2
[00000000] *** left: 3
[00000000] *** done
// Run test in background, then both echos without waiting.
test("left"), echo("right"); echo("done");
[00000000] *** left: 1
[00000000] *** right
[00000000] *** left: 2
[00000000] *** done
[00000000] *** left: 3
```

That’s about all there is to say about these operators. Although they’re rather simple, they are really powerful and enables you to include parallelism anywhere at no syntactical cost.

14.2 Detach

The `Control.detach` function backgrounds the execution of its argument. Its behavior is the same as the comma (,) operator, except that the execution is completely detached, and not waited for at the end of the scope.

urbiscript
Session

```
function test()
{
    // Wait for one second, and echo "foo".
    detach({sleep(1s); echo("foo")});
}
test();
echo("Not blocked");
[00000000] Job<shell_4>
[00000000] *** Not blocked
sleep(2s);
echo("End of sleep");
[00001000] *** foo
[00002000] *** End of sleep
```

14.3 Tags for parallel control flows

A [Tag](#) is a multipurpose code execution control and instrumentation feature. Any chunk of code can be tagged, by preceding it with a tag and a colon (:). Tag can be created with `Tag.new(name)`. Naming tags is optional, yet it's a good idea since it will be used for many features. The example below illustrates how to tag chunks of code.

```
// Create a new tag
var mytag = Tag.new("name");
[00000000] Tag<name>
// Tag the evaluation of 42
mytag: 42;
[00000000] 42
// Tag the evaluation of a block.
mytag: { "foo"; 51 };
[00000000] 51
// Tag a function call.
mytag: echo("tagged");
[00000000] *** tagged
```

urbiscript
Session

You can use tags that were not declared previously, they will be created implicitly (see below). However, this is not recommended since tags will be created in a global scope, the `Tag` object. This feature can be used when inputting test code in the top level to avoid bothering to declare each tag, yet it is considered poor practice in regular code.

```
// Since mytag is not declared, this will first do:
// var Tag.mytag = Tag.new("mytag");
mytag : 42;
[00000000] 42
```

urbiscript
Session

So you can tag code, yet what's the use? One of the primary purpose of tags is to be able to control the execution of code running in parallel. Tags have a few control methods (see [Tag](#)):

freeze Suspend execution of all tagged code.

unfreeze Resume execution of previously frozen code.

stop Stop the execution of the tagged code. The flows of execution that where stopped jump immediately at the end of the tagged block.

block Block the execution of the tagged code, that is:

- Stop it.
- When an execution flow encounters the tagged block, it simply skips it.

You can think of `block` like a permanent `stop`.

unblock Stop blocking the tagged code.

The three following examples illustrate these features.

```
// Launch in background (using the comma) code that prints "ping"
// every second. Tag it to keep control over it.
mytag:
  every (1s)
    echo("ping"),
  sleep(2.5s);
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
// Suspend execution
mytag.freeze;
// No printing anymore
```

urbiscript
Session

```
sleep(1s);
// Resume execution
mytag.unfreeze;
sleep(1s);
[00007000] *** ping
```

urbiscript
Session

```
// Now, we print out a message when we get out of the tag.
{
    mytag:
        every (1s)
            echo("ping");
    // Execution flow jumps here if mytag is stopped.
    echo("Background job stopped")|
},
sleep(2.5s);
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
// Stop the tag
mytag.stop;
[00002500] *** Background job stopped
// Our background job finished.
// Unfreezing the tag has no effect.
mytag.unfreeze;
```

urbiscript
Session

```
// Now, print out a message when we get out of the tag.
loop
{
    echo("ping"); sleep(1s);
    mytag: { echo("pong"); sleep(1s); };
},
sleep(3.5s);
[00000000] *** ping
[00001000] *** pong
[00002000] *** ping
[00003000] *** pong

// Block printing of pong.
mytag.block;
sleep(3s);

// The second half of the while isn't executed anymore.
[00004000] *** ping
[00005000] *** ping
[00006000] *** ping

// Reactivate pong
mytag.unblock;
sleep(3.5s);
[00008000] *** pong
[00009000] *** ping
[00010000] *** pong
[00011000] *** ping
```

14.4 Advanced example with parallelism and tags

In this section, we implement a more advanced example with parallelism.

The listing below presents how to implement a `timeOut` function, that takes code to execute and a timeout as arguments. It executes the code, and returns its value. However, if the code execution takes longer than the given timeout, it aborts it, prints "Timeout!" and returns `void`. In this example, we use:

- Lazy evaluation, since we want to delay the execution of the given code, to keep control on it.
- Concurrency operators, to launch a timeout job in background.

```
// timeout (Code, Duration).
function timeOut
{
    // In background, launch a timeout job that waits
    // for the given duration before aborting the function.
    // call.evalArgAt(1) is the second argument, the duration.
    {
        sleep(call.evalArgAt(1));
        echo("Timeout!");
        return;
    },
    // Run the Code and return its value.
    return call.evalArgAt(0);
} |;
timeOut({sleep(1s); echo("On time"); 42}, 2s);
[00000000] *** On time
[00000000] 42
timeOut({sleep(2s); echo("On time"); 42}, 1s);
[00000000] *** Timeout!
```

urbiscript
Session

Chapter 15

Event-based Programming

When dealing with highly interactive agent programming, sequential programming is inconvenient. We want to react to external, random events, not execute code linearly with a predefined flow. `urbiscript` has a strong support for event-based programming.

15.1 Watchdog constructs

The first construct we will study uses the `at` keyword. Given a condition and a statement, `at` will evaluate the statement each time the condition *becomes* true. That is, when a rising edge occurs on the condition.

```
var x = 0;
[00000000] 0
at (x > 5)
  echo("ping");
x = 5;
[00000000] 5
// This triggers the event.
x = 6;
[00000000] 6
[00000000] *** ping
// Does not trigger, since the condition is already true.
x = 7;
[00000000] 7
// The condition becomes false here.
x = 3;
[00000000] 3

x = 10;
[00000000] 10
[00000000] *** ping
```

urbiscript
Session

An `onleave` block can be appended to execute an expression when the expression *becomes* false — that is, on falling edges.

```
var x = false;
[00000000] false
at (x)
  echo("x")
onleave
  echo("!x");
x = true;
[00000000] true
[00000000] *** x
x = false;
[00000000] false
[00000000] *** !x
```

urbiscript
Session

See [Section 23.10.1](#) for more details on `at` statements.

The `whenever` construct is similar to `at`, except the expression evaluation is systematically restarted when it finishes as long as the condition stands true.

urbiscript
Session

```
var x = false;
[00000000] false
whenever (x)
{
    echo("ping");
    sleep(1s);
};

x = true;
[00000000] true
sleep(3s);
// Whenever keeps triggering
[00000000] *** ping
[00001000] *** ping
[00002000] *** ping
x = false;
[00002000] false
// Whenever stops triggering
```

Just like `at` has `onleave`, `whenever` has `else`: the given expression is evaluated as long as the condition is false.

urbiscript
Session

```
var x = false;
[00002000] false
whenever (x)
{
    echo("ping");
    sleep(1s);
}
else
{
    echo("pong");
    sleep(1s);
};
sleep (3s);
[00000000] *** pong
[00001000] *** pong
[00002000] *** pong
x = true;
[00003000] true
sleep (3s);
[00003000] *** ping
[00004000] *** ping
[00005000] *** ping
x = false;
[00006000] false
sleep (2s);
[00006000] *** pong
[00007000] *** pong
```

15.2 Events

In addition to monitoring an expression with a watchdog, urbiscript enables you to define events that can be caught with the `at` and `whenever` constructs we saw earlier. You can create events by instantiating the `Event` prototype. They can then be emitted with the `!` keyword.

15.2.1 Emitting Events

urbiscript
Session

```

var myEvent = Event.new;
[00000000] Event_Oxb5579008
at (myEvent?)
  echo("ping");
myEvent!;
[00000000] *** ping
// events work well with parallelism
myEvent! & myEvent!;
[00000000] *** ping
[00000000] *** ping

```

Both `at` and `whenever` have the same behavior on punctual events. However, if you emit an event for a given duration, `whenever` will keep triggering for this duration, contrary to `at`.

```

var myEvent = Event.new;
[00000000] Event_Oxb558a588
whenever (myEvent?)
{
  echo("ping (whenever)") |
  sleep(200ms)
};
at (myEvent?)
{
  echo("ping (at)") |
  sleep(200ms)
};
// Emit myEvent for .3 second.
myEvent! ~ 300ms;
[00000000] *** ping (whenever)
[00000100] *** ping (whenever)
[00000000] *** ping (at)

```

urbiscript
Session

15.2.2 Emitting events with a payload

Events behave very much like “channels”: listeners use `at` or `whenever`, and producers use `!`. Fortunately, the messages can include a *payload*, i.e., something sent in the “message”. The Event then behaves very much like an identifier of the message type. To send/catch the payload, just pass arguments to `!` and `?`:

```

var event = Event.new;
[00000000] Event_Ox0

at (event?(var payload))
  echo("received: " + payload)
onleave
  echo("had received: " + payload);

event!(1);
[00000008] *** received: 1
[00000009] *** had received: 1

event!(["string", 124]);
[00000010] *** received: ["string", 124]
[00000011] *** had received: ["string", 124]

```

urbiscript
Session

Like functions, events have an *arity*, i.e., they depend on the number of arguments: `at (event?(arg))` will only match emissions whose payload contain exactly one argument, i.e., `event!(arg)`.

```

// Too many arguments.
event!(1, 2);

// Not enough arguments.
event!;

```

urbiscript
Session

```
event!();
```

Event handlers that do not specify their arity (i.e., without parentheses) match event emissions of any arity.

urbiscript
Session

```
at (event?)
  echo("received an event")
onleave
  echo("had received an event");

event!;
[00000014] *** received an event
[00000015] *** had received an event

event!(1);
[00000016] *** received: 1
[00000017] *** had received: 1
[00000018] *** received an event
[00000019] *** had received an event

event!(1, 2);
[00000020] *** received an event
[00000021] *** had received an event
```

Actually, the feature is much more powerful than this: full pattern matching applies, as with the `switch/case` construct.

urbiscript
Session

```
var e = Event.new();

at (e?)
  echo("e");

at (e?(var x))
  echo("e(x)");

at (e?(1))
  echo("e(1)");

at (e?(var x) if x.isA(Float) && x % 2)
  echo("e(odd)");

// Payload must be a list of three members, the first two being 1 and 2, and
// the third one being greater than 2, when converted as a Float.
at (e?([1, 2, var x]) if 2 < x.asFloat)
  echo("e([1, 2, x = %s])" % x);

e!;
[00000845] *** e

e!(0);
[00011902] *** e
[00011902] *** e(x)

e!(1);
[00023327] *** e
[00023327] *** e(x)
[00023327] *** e(1)
[00023327] *** e(odd)

e!([1, 2, 1]);
[00024327] *** e
[00024327] *** e(x)

e!([1, 2, 3]);
[00025327] *** e
```

```
[00025327] *** e(x)
[00025327] *** e([1, 2, x = 3])
e!([1, 2, "4"]);
[00026327] *** e
[00026327] *** e(x)
[00026327] *** e([1, 2, x = 4])
```


Chapter 16

Urbi for ROS Users

This chapter extends the [ROS official tutorials](#)¹. Be sure to complete this tutorial before reading this document.

16.1 Communication on topics

First we will take back examples about topics; make sure that talker and listener in the ‘beginner_tutorial’ package are compiled. You can recompile it with the following command:

```
$ rosmake beginner_tutorial
```

Shell
Session

16.1.1 Starting a process from Urbi

To communicate with ROS components, you need to launch them. You can do it by hand, or ask Urbi to do it for you. To launch new processes through Urbi, we will use the class [Process](#).

Let’s say we want to start `roscore`, and the talker of the beginner tutorial. Open an Urbi shell by typing the command ‘`rlwrap urbi -i`’. Here `rlwrap` makes ‘`urbi -i`’ acts like a shell prompt, with features like line editing, history, …

```
var core = Process.new("roscore", []);
[00000001] Process roscore
var talker = Process.new("rosrun", ["beginner_tutorial", "talker"]);
[00000002] Process rosrun
core.run;
talker.run;
```

urbiscript
Session

At this point, the processes are launched. The first argument of `Process.new` is the name of the command to launch, the second is a list of arguments.

Then you can check the status of the processes, get their stdout/stderr buffers, kill them in `urbiscript` (see [Process](#)).

16.1.2 Listening to Topics

First you need to make sure that `roscore` is running, and the ROS module is loaded correctly:

```
Global.hasLocalSlot("Ros");
[00016931] true
```

urbiscript
Session

Then we can get the list of launched nodes:

```
Ros.nodes;
```

urbiscript
Session

This returns a [Dictionary](#) with the name of the node as key, and a dictionary with topics subscribed, topics advertised, topics advertised as value.

We can check that our talker is registered, and on which channel it advertises:

```
urbiscript
Session
// Get the structure.
// ";" is an idiom to discard the display of the return value.
var nodes = Ros.nodes|;

// List of nodes (keys).
nodes.keys;
[00000002] ["/rosout", "/urbi_1273060422295250703", "/talker"]

// Details of the node "talker".
nodes["talker"]["publish"];
[00000003] ["/rosout", "/chatter"]
```

Here we see that this node advertises '/rosout' and '/chatter'. Let's subscribe to '/chatter':

```
urbiscript
Session
// Initialize the subscription object.
var chatter = Ros.Topic.new("/chatter")|;
// Subscribe.
chatter.subscribe;
// This is the way we are called on new message.
var chatTag = Tag.new|;
chatTag: at (chatter.onMessage?(var e))
// This will be executed on each message.
echo(e);
```

In this code, `e` is a Dictionary that follows the structure of the ROS message. Here is an example of what this code produces:

```
urbiscript
Session
[00000004] *** ["data" => "Hello there! This is message [4]"]
[00000005] *** ["data" => "Hello there! This is message [5]"]
[00000006] *** ["data" => "Hello there! This is message [6]"]
```

We can also get a template for the message structure on this channel with:

```
urbiscript
Session
chatter.structure;
[00000007] ["data" => ""]
```

To stop temporarily the `Global.echo`, we take advantages of tags (Section 14.3), by doing `chatTag.freeze`. Same thing goes with unfreeze. Of course you could also call `chatter.unsubscribe`, which unsubscribes you completely from this channel.

16.1.3 Advertising on Topics

To advertise a topic, this is roughly the same procedure.

16.1.3.1 Simple Talker

Here is a quick example:

```
urbiscript
Session
// Initialize our object.
var talker = Ros.Topic.new("/chatter")|;
// Advertise (providing the ROS Type of this topic).
talker.advertise("std_msgs/String");

// Get a template of our structure.
var msg = talker.structure.new;
msg["data"] = "Hello ROS world";
talker << msg;
```

We have just sent our first message to ROS, here if you launch the chatter, you will be able to get the message we have just sent.

The `<<` operator is an convenient alias for `Ros.Topic.publish`.

¹<http://www.ros.org/wiki/ROS/Tutorials>

16.1.3.2 Turtle Simulation

Now we are going to move the turtle with Urbi. First let's launch the turtle node:

```
var turtle = Process.new("rosrun", ["turtlesim", "turtlesim_node"]);|
turtle.run;
```

urbiscript
Session

`Ros.topics` shows that this turtle subscribes to a topic '/turtle1/command_velocity'.

Let's advertise on it:

```
var velocity = Ros.Topic.new("/turtle1/command_velocity")|;
velocity.advertise("turtlesim/Velocity");
velocity.structure;
[00000001] ["linear" => 0, "angular" => 0]
```

urbiscript
Session

Now we want to have it moving in circle with a small sinusoid wave. This goes in two step. First, we set up the infrastructure so that changes in Urbi are seamlessly published in ROS.

```
// Get our template structure.
var m = velocity.structure.new|;
m["linear"] = 0.8 |;
var angular = 0 |;
// Every time angular is changed, we send a message.
at (angular->changed?)
{
    m["angular"] = angular;
    velocity << m
};
```

urbiscript
Session

In the future Urbi will provide helping functions to spare the user from the need to perform this “binding”. But once this binding done, all the features of urbiscript can be used transparently.

For instance we can assign a sinusoidal trajectory to 'angular', which results in the screen-shot on the right-hand side.

```
// For 20 seconds, bind "angular" to a sine.
timeout (20s)
angular = 0.3 sin: 2s ampli: 2;
```



Every time `angular` is changed, a new message is sent on the Topic '/turtle1/command_velocity', thus updating the position of the turtle. After 20 seconds the command is stopped.

Alternatively, `Tags` could have been used to get more control over the trajectory:

```
// A Tag to control the following endless statement.
var angTag = Tag.new|;

angTag:
    // Bind "angular" to a trajectory.
    // Put in background thanks to ",", since this statement is never ending.
    angular = 0.3 sin: 2s ampli: 2,
    // Leave 20 seconds to the turtle...
    sleep(20s);
```

urbiscript
Session

```
// before freezing it.
angTag.freeze;
```

We won't cover this code in details, but the general principle is that `angular` is updated every 20ms with the values of a sinusoid wave trajectory with 0.3 as average value, 2 seconds for the period and 2 for the amplitude. See [TrajectoryGenerator](#) for more information. After 20 seconds the tag is frozen, pausing the trajectory generation and the `at`.

16.2 Using Services

Services work the same way topics do, with minor differences.

Let's take back the turtle simulation example ([Section 16.1.3.2](#)). Then we can list the available services, and filter out loggers:

```
urbiscript
Session
var logger = Regexp.new("(get|set)_logger") |;
var services = Ros.services.keys |;
for (var s in services)
  if (s not in logger)
    echo(s);
[00000001] *** "/clear"
[00000001] *** "/kill"
[00000001] *** "/turtle1/teleport_absolute"
[00000001] *** "/turtle1/teleport_relative"
[00000001] *** "/turtle1/set_pen"
[00000001] *** "/reset"
[00000001] *** "/spawn"
```

The `closure` construct allows us to keep access to the local variables, here `logger`.

Now there is a service called '`/spawn`'; to initialize it:

```
urbiscript
Session
var spawn = Ros.Service.new("/spawn", false) |;
waituntil(spawn.initialized);
```

The `new` function takes the service name as first argument, and as second argument whether the connection should be kept alive.

Since the creation of this object checks the service name, you should wait until `initialized` is true to use this service. You can also see the structure of the request with `spawn.reqStruct`, and the structure of the response with `spawn.resStruct`.

Now let's spawn a turtle called Jenny, at position (4, 4).

```
urbiscript
Session
var req = spawn.reqStruct.new |;
req["x"] = 4 |
req["y"] = 4 |
req["name"] = "Jenny" |;
spawn.request(req);
[00000001] ["name" => "Jenny"]
```

16.3 Image Publisher from ROS to Urbi

This section will use topics manipulation with advertising and subscription. Be sure to understand these topics before doing this tutorial.

Requirements You have to finish the image Publisher/Subscriber tutorial (http://www.ros.org/wiki/image_transport/Tutorials) before doing this tutorial.

First, we will make a ROS Publisher and subscribe to it with Urbi. Make sure that Publisher 'learning_image_transport' package is compiled:

```
Shell
Session
$ rosmake learning_image_transport
```

We will also run `urbi` with a network connection opened (e.g., on port 54000) to allow `urbi-image` (Section 22.4) to connect to it.

```
$ urbi --host=127.0.0.1 --port=54000 -- -f
```

Shell Session

Also, you have to run `roscore` to communicate with ROS.

```
var core = Process.new("roscore", []);
[00000001] Process roscore
core.run;
```

urbiscript Session

Run the Publisher The Publisher is a process that will send a image and wait for a Subscriber to get it.

```
// In this example the image is in the current directory.
var publisher =
  Process.new("rosrun",
    ["learning_image_transport", "my_publisher", "test.jpg"]);
[00000002] Process rosrun
publisher.run;
```

urbiscript Session

Using a camera to display By default, `urbi-image` displays the images that are available via the `camera` device (see Section 22.4). To simplify the setup, let's define a pseudo `camera` which will store the data received:

```
class Global.camera: Loadable
{
  // A variable to store image data.
  UVar.new(this, "val");
  val = 0;
};
```

urbiscript Session

Subscribe to the topic Now, our Publisher is running and we have a camera waiting for data. All we need to do is connecting to the Publisher with a topic, the Subscriber.

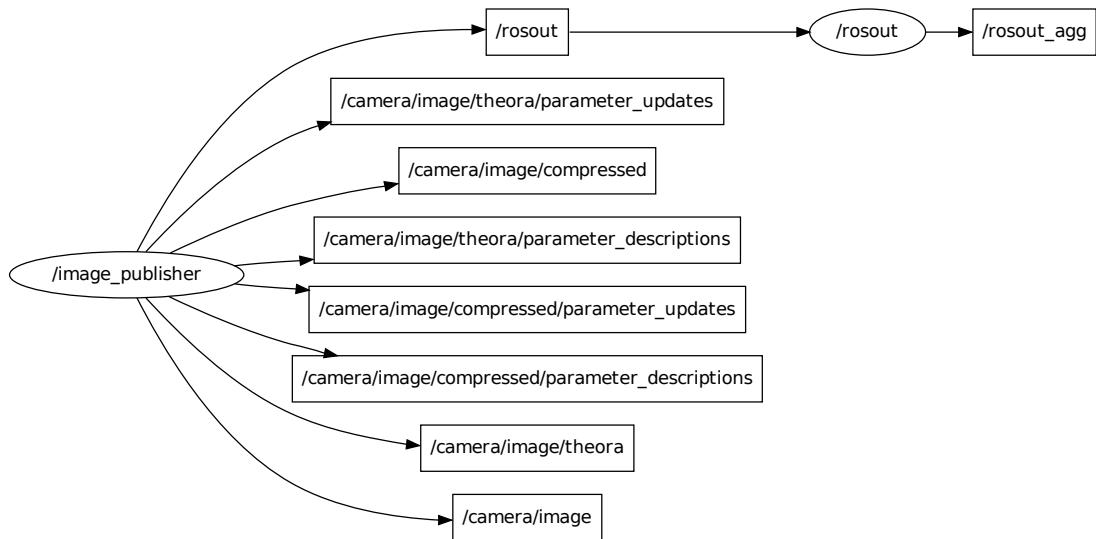


Figure 16.1: Output from `rxgraph`

Have a look at the different topics created by the Publisher, for instance by running `rxgraph`, which generates the graph in Figure 16.1. As you can see, seven topics are available for the camera. We will use the '`/camera/image/compressed`' topic for this example. For further

information about the image format in ROS see http://www.ros.org/doc/api/sensor_msgs/html/msg/CompressedImage.html.

urbiscript
Session

```
var cameraTopic = Ros.Topic.new("/camera/image/compressed")|;
at (cameraTopic.onMessage?(var imgMsg))
{
    // Converting the ROS image to Urbi format.
    imgMsg["data"].keywords = imgMsg["format"]|
    // We can now store the data into camera.
    if (!camera.val)
        echo("Image well received. Store the image into the camera") |
        camera.val = imgMsg["data"];
},
// Waiting for the "publisher" Process to be set up.
sleep(2s);
cameraTopic.subscribe;
```

We are now connected and ready to display.

urbiscript
Session

```
[00000003] *** Image well received. Store the image into the camera
```

In a new terminal run `urbi-image`:

Shell
Session

```
$ urbi-image
Monitor created window 62914561
***Frame rate: 5.000000 fps***
```

You have now your image displayed in a window.

16.4 Image Subscriber from Urbi to ROS

Now, we want to send images to ROS using a Urbi Publisher. Make sure `roscore` is running and ‘`learning_image_transport`’ package is compiled.

Run the Subscriber The basic Subscriber in the ‘`learning_image_transport`’ package is expecting a ‘`/camera/image`’ topic. To avoid modifying the Subscriber code in ROS, we will simply ask to the Subscriber topic to accept ‘`/camera/image/compressed`’ topics.

urbiscript
Session

```
var subscriber =
    Process.new("rosrun",
        ["learning_image_transport", "my_subscriber",
        "_image_transport:=compressed"]);
[00037651] Process rosrun
subscriber.run;
```

Publishing images with Urbi The ‘`sensor_msgs/CompressedImage`’ message format provides a structure that requires a few changes.

urbiscript
Session

```
// File.new("...").content returns a Binary.
var urbiImage = File.new("test.jpg").content|;
urbiImage.keywords = "jpeg"|

var publisher = Ros.Topic.new("/camera/image/compressed")|;
// Advertising the type of message used.
publisher.advertise("sensor_msgs/CompressedImage");

var rosImg = publisher.structure.new|;
// The rosImg is a dictionary containing a Binary and a String.
rosImg["data"] = urbiImage|;
rosImg["format"] = "jpeg"|;
```

This message contains more fields but you need only these two to send an image.
Now, you just have to publish the image.

```
// Publishing at regular intervals.  
every (500ms)  
{  
    publisher << img;  
},
```

urbiscript
Session

Communication is done, the image should be displayed.

16.5 Remote communication

We have worked with a `roscore` running on the machine as the ROS processes but the purpose of using ROS with Urbi is to communicate with a remote machine. All you need is to setup your network configuration to avoid unexpected behaviors (see [NetworkSetup²](#)).

Make sure the ROS environment variables are well set, especially `ROS_URI`, `ROS_HOSTNAME`, `ROS_IP`.

See [Tutorials/MultipleMachines³](#) for additional information.

Try our tutorials remotely to check if the connection is set correctly.

To go further... Please see the Urbi/ROS Reference Manual, [Chapter 25](#).

²<http://www.ros.org/wiki/ROS/NetworkSetup>

³<http://www.ros.org/wiki/ROS/Tutorials/MultipleMachines>

Part III

Guidelines and Cook Books

About This Part

This part contains guides to some specific aspects of Urbi SDK.

Chapter 17 — Installation

Complete instructions on how to install Urbi SDK.

Chapter 18 — Frequently Asked Questions

Some answers to common questions.

Chapter 19 — Urbi Guideline

Based on our own experience, and code that users have submitted to us, we suggest a programming guideline for Urbi SDK.

Chapter 20 — Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2.

Chapter 21 — Building Urbi SDK

Building Urbi SDK from the sources. How to install it, how to check it and so forth.

Chapter 17

Installation

17.1 Download

Various pre-compiled packages are provided. They are named

`'urbi-sdk-version-arch-os-compiler.ext'`

where

version specifies the exact revision of Urbi that you are using. It can be simple, 2.0, or more complex, 2.0-beta3-137-g28f8880. In that case,

2.0 is the version of the Urbi Kernel,

beta3 designates the third pre-release,

137 is the number of changes since beta3 (not counting changes in sub-packages),

g28f8880 is a version control identifier, used internally to track the exact version that is being tested by our users.

arch describes the architecture, the CPU: ARM, ppc, or x86.

os is the operating system: linux for GNU/Linux, osx for Mac OS X, or windows for Microsoft Windows.

compiler is the tool chain used to compile the programs: gcc4 for the GNU Compiler Collection 4.x, vcxx2005 for Microsoft Visual C++ 2005, vcxx2008 for Microsoft Visual C++ 2008.

ext is the package format extension. For Unix architectures, tar.bz2; uncompress them with `tar xf tarfile`. For Windows hosts, we provide zip files ('*.zip') for both “release” and “debug” flavors, and installers (for instance ‘*.exe’). You are encouraged to use the installers, since in addition to installing headers and libraries, they also install Visual C++ Wizards to create UObjects, they take care of installing the Visual Runtime if needed, and they install Gostai Console and Gostai Editor.

17.2 Install & Check

The package is *relocatable*, i.e., it does not need to be put at a specific location, nor does it need special environment variables to be set. It is not necessary to be a super-user to install it. The *root* of the package, denoted by *urbi-root* hereafter, is the absolute name of the directory which contains the package.

After the install, the quickest way to test your installation is to run the various programs.

17.2.1 GNU/Linux and Mac OS X

Decompress the package where you want to install it. If *urbi-sdk-2.x* denotes the version of Urbi SDK you downloaded (say, *urbi-sdk-2.x* is ‘*urbi-sdk-2.3-linux-x86-gcc4*’), run something like:

Shell Session

```
$ rm -rf urbi-root
$ cd /tmp
$ tar xf path-to/urbi-sdk-2.x.tar.bz2
$ mv urbi-sdk-2.x urbi-root
```

This directory, *urbi-root*, should contain ‘bin’, ‘FAQ.txt’ and so forth. Do not move things around inside this directory. In order to have an easy access to the Urbi programs, set up your PATH:

Shell Session

```
$ export PATH="urbi-root/bin:$PATH"
```

Shell Session

```
# Check that urbi is properly set up.
$ urbi --version

# Check that urbi-launch is properly installed.
$ urbi-launch --version
# Check that urbi-launch can find its dependencies.
$ urbi-launch -- --version

# Check that Urbi can compute.
$ urbi -e '1+2*3; shutdown;'
[00000175] 7
```

17.2.2 Windows

Decompress the zip file wherever you want or execute the installer.

Execute the script ‘*urbi.bat*’, located at the root of the uncompressed package. It should open a terminal with an interactive Urbi session.

Cygwin Issues

Inputs and outputs of windows native application are buffered under Cygwin. Thus, either running the interactive mode of Urbi or watching the output of the server under Cygwin is not recommended.

Chapter 18

Frequently Asked Questions

18.1 Build Issues

18.1.1 Complaints about ‘+=’

Although we tried to avoid it, there might still be shell scripts where we use ‘+=’, which Ash (aka, dash and sash) does not support. Please, use `bash` or `zsh` instead of Ash as `/bin/sh`.

18.1.2 error: ‘<anonymous>’ is used uninitialized in this function

If you encounter this error:

```
cc1plus: warnings being treated as errors
parser/ugrammar.hh: In member function \
  'void yy::parser::yypush_(const char*, int, yy::parser::symbol_type&)' :
parser/ugrammar.hh:1240: error: '<anonymous>' is used uninitialized \
  in this function
parser/ugrammar.cc:1305: note: '<anonymous>' was declared here
parser/ugrammar.hh: In member function \
  'void yy::parser::yypush_(const char*, yy::parser::stack_symbol_type&)' :
parser/ugrammar.hh:1240: error: '<anonymous>' is used uninitialized \
  in this function
parser/ugrammar.cc:1475: note: '<anonymous>' was declared here
```

then you found a problem that we don't know how to resolved currently. Downgrade from GCC-4.4 to GCC-4.3.

18.1.3 AM_LANGINFO_CODESET

If at bootstrap you have something like:

```
configure:12176: error: possibly undefined macro: AM_LANGINFO_CODESET
  If this token and others are legitimate, please use m4_pattern_allow.
  See the Autoconf documentation.
configure:12246: error: possibly undefined macro: gl_GLIBC21
```

it probably means your Automake installation is incomplete. See the Automake item in [Section 21.1](#).

18.1.4 configure: error: The Java VM java failed

If you experience the following failure:

```
checking if java works...
configure: error: The Java VM java failed
  (see config.log, check the CLASSPATH?)
```

and if you looked at ‘`config.log`’, you should find something like:

```

Exception in thread "main" java.lang.NoClassDefFoundError: Test
Caused by: java.lang.ClassNotFoundException: Test
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
Could not find the main class: Test. Program will exit.

```

You might be trying to compile Urbi SDK from a directory with non-ASCII characters in its full name (for instance ‘/home/jessy/Téléchargements/Sources’). In that case the JVM fails to decode properly the path, and `configure` fails with the above message.

Move your sources elsewhere, with only plain ASCII characters, such as ‘/home/jessy/Sources’.

18.1.5 ‘make check’ fails

Be sure to read [Section 21.9](#). In particular, run ‘`make check`’ several times (see [Section 21.9](#) to know why). If the failures remain, please submit the ‘`test-suite.log`’ file(s) (see [Section 18.5.3](#)).

18.1.6 check: error: Unable to load native library: libjava.jnilib

If you experience the following failure on Mac OS X:

```

$ urbi-launch-java --port-file server.port tests/all/All.jar
Error occurred during initialization of VM
Unable to load native library: libjava.jnilib

```

then you have something in the `DYLD_LIBRARY_PATH` that annoys the Java VM. Although we have not pinpointed it exactly, it seems to be a problem in our `libjpeg`: first run ‘`make -C sdk-remote/jpeg clean`’ then ‘`make -S sdk-remote/jpeg`’.

18.2 Troubleshooting

18.2.1 error while loading shared libraries: libport.so

If on GNU/Linux you get an error such as:

Shell Session	<pre> urbi-sdk/2.7.1 \$./bin/urbi ./bin/urbi: error while loading shared libraries: libport.so: \ cannot open shared object file: No such file or directory </pre>
---------------	---

then check that ‘/proc’ is properly mounted. To make Urbi SDK relocatable, executables and libraries use a relative path to their peers. To resolve these paths into absolute paths, the loader needs to know where the program is located, a feature provided by ‘/proc’. If for instance you run Urbi SDK in a chrooted environment, then it is possible that you forgot to mount ‘/proc’. The traditional `ps` utility also needs ‘/proc’ to be mounted, so running it would also help checking if the setup is complete.

18.2.2 Error 1723: “A DLL required for this install to complete could not be run.”

This error is raised when you try to install a program like `vcredist-x86.exe`. This program uses the “Windows Installer” which is probably outdated on your system.

To fix this problem, update the “Windows Installer” and re-start the installation of `vcredist` which should no longer fail.

18.2.3 When executing a program, the message “The system cannot execute the specified program.” is raised.

This library is necessary to start running any application. Run ‘vcredist-x86.exe’ to install the missing libraries.

If you have used the Urbi SDK installer, it is ‘vcredist-x86.exe’ in your install directory. Otherwise download it from the Microsoft web site. Be sure to get the one corresponding to the right Visual C++ version.

18.2.4 When executing a program, the message “This application has failed to start” is raised.

Same answer as [Section 18.2.3](#).

18.2.5 The server dies with “stack exhaustion”

Your program might be deeply recursive, or use large temporary objects. Use ‘--stack-size’ to augment the stack size, see [Section 22.3](#).

Note that one stack is allocated per “light thread”. This can explain why programs that heavily rely on concurrency might succeed where sequential programs can fail. For instance the following program is very likely to quickly exhaust the (single) stack.

```
function consume (var num)
{
    if (num)
        consume(num - 1) | consume(num - 1)
}|;
consume (512);
```

urbiscript
Session

But if you use & instead of |, then each recursive call to `consume` will be spawn with a fresh stack, and therefore none will run out of stack space:

```
function consume (var num)
{
    if (num)
        consume(num - 1) & consume(num - 1)
}|;
consume (512);
```

urbiscript
Session

However your machine will run out of resources: this heavily concurrent program aims at creating no less than 2^{513} threads, about 2.68×10^{156} (a 156-digit long number, by far larger than the number of atoms in the observable universe, estimated to 10^{80}).

18.2.6 ‘myuobject: file not found’. What can I do?

If `urbi-launch` (or `urbi`) fails to load an UObject (a shared library or DLL) although the file exists, then the most probable cause is an undefined symbol in your shared library.

18.2.6.1 Getting a better diagnostic

First, set the `GD_LEVEL` environment variable (see [Section 22.1.2](#)) to some high level, say `DUMP`, to log messages from `urbi-launch`. You might notice that your library is not exactly where you thought `urbi-launch` was looking at.

18.2.6.2 GNU/Linux

A libltdl quirk prevents us from displaying a more accurate error message. You can use a tool named `ltrace` to obtain the exact error message. Ltrace is a standard package on most Linux distributions. Run it with ‘`ltrace -C -s 1024 urbi-launch ...`’, and look for lines

containing ‘dlsym’ in the output. One will contain the exact message that occurred while trying to load your shared library.

It is also useful to use ldd to check that the dependencies of your object are correct. See the documentation of ldd on your machine (‘man ldd’). The following run is successful: every request (left-hand side of =>) is satisfied (by the file shown on the right-hand side).

Shell Session

```
$ all.so
    linux-gate.so.1 => (0xb7fe8000)
    libstdc++.so.6 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libstdc++.so.6 (0xb7eba000)
    libm.so.6 => /lib/libm.so.6 (0xb7e94000)
    libc.so.6 => /lib/libc.so.6 (0xb7d51000)
    libgcc_s.so.1 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libgcc_s.so.1 (0xb7d35000)
    /lib/ld-linux.so.2 (0xb7fe9000)
```

The following run shows a broken dependency.

Shell Session

```
# A simple C++ program.
$ echo 'int main() {}' >foo.cc

# Compile it, and depend on the libport shared library.
$ g++ foo.cc -Lurbi-root/gostai/lib -lport -o foo

# Run it.
$ ./foo
./foo: error while loading shared libraries: \
  libport.so: cannot open shared object file: No such file or directory

# See that ldd is unhappy.
$ ldd foo
    linux-gate.so.1 => (0xb7fa4000)
    libport.so => not found
    libstdc++.so.6 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libstdc++.so.6 (0xb7eae000)
    libm.so.6 => /lib/libm.so.6 (0xb7e88000)
    libgcc_s.so.1 => \
        /usr/lib/gcc/i686-pc-linux-gnu/4.4.1/libgcc_s.so.1 (0xb7e6c000)
    libc.so.6 => /lib/libc.so.6 (0xb7d29000)
    /lib/ld-linux.so.2 (0xb7fa5000)
```

Notice the ‘not found’ message. The shared object could not be loaded because it is not found in the *runtime path*, which is the list of directories where the system looks for shared objects to be loaded when running a program.

You may extend your LD_LIBRARY_PATH to include the missing directory.

Shell Session

```
$ export LD_LIBRARY_PATH=urbi-root/gostai/lib:$LD_LIBRARY_PATH
# Run it.
$ ./foo
```

18.2.6.3 Mac OS X

Set the DYLD_PRINT_LIBRARIES environment variable to 1 to make the shared library loader report the libraries it loads on the standard error stream.

Use otool to check whether a shared object “finds” all its dependencies.

Shell Session

```
$ otool -L all.so
all.so:
    /usr/lib/libstdc++.6.dylib \
        (compatibility version 7.0.0, current version 7.4.0)
    /usr/lib/libgcc_s.1.dylib \
        (compatibility version 1.0.0, current version 1.0.0)
    /usr/lib/libSystem.B.dylib \
```

```
(compatibility version 1.0.0, current version 111.1.4)
```

The following run shows a broken dependency.

```
# A simple C++ program.
$ echo 'int main() {}' >foo.cc

# Compile it, and depend on the libport shared library.
$ g++ foo.cc -Lurbi-root/gostai/lib -lport -o foo

# Run it.
$ ./foo
dyld: Library not loaded: @loader_path/libport.dylib
Referenced from: /private/tmp./foo
Reason: image not found

# See that otool is unhappy.
$ otool -L ./foo
./foo:
    @loader_path/libport.dylib \
        (compatibility version 0.0.0, current version 0.0.0)
    /usr/lib/libstdc++.6.dylib \
        (compatibility version 7.0.0, current version 7.4.0)
    /usr/lib/libgcc_s.1.dylib \
        (compatibility version 1.0.0, current version 1.0.0)
    /usr/lib/libSystem.B.dylib \
        (compatibility version 1.0.0, current version 111.1.5)
```

Shell Session

The fact that the ‘libport.dylib’ was not found shows by the unresolved relative runtime-path: ‘@loader_path’ still shows. Use DYLD_LIBRARY_PATH to specify additional directories where the system should look for runtime dependencies.

```
$ DYLD_PRINT_LIBRARIES=1 \
DYLD_LIBRARY_PATH=urbi-root/lib:$DYLD_LIBRARY_PATH \
./foo
dyld: loaded: /private/tmp./foo
dyld: loaded: urbi-root/lib/libport.dylib
dyld: loaded: /usr/lib/libstdc++.6.dylib
dyld: loaded: /usr/lib/libgcc_s.1.dylib
dyld: loaded: /usr/lib/libSystem.B.dylib
dyld: loaded: urbi-root/lib/libboost_filesystem-mt.dylib
dyld: loaded: urbi-root/lib/libboost_signals-mt.dylib
dyld: loaded: urbi-root/lib/libboost_system-mt.dylib
dyld: loaded: urbi-root/lib/libboost_thread-mt.dylib
dyld: loaded: /usr/lib/system/libmathCommon.A.dylib
$
```

Shell Session

18.2.6.4 Windows

If you are running Cygwin, then have a look at the following section, which uses some of its tools.

A specific constraint, for which currently we do not have nice solutions, is that when Windows loads a DLL, it looks for all its dependencies (i.e., other DLL that are needed) in the directory from which the program was run, or in the PATH. There is no way, that we are aware of, to embed in a DLL the information about where the dependencies are. When trying to load a DLL with missing dependencies, say ‘foo.dll’, the error message will be something like “can’t open the module”, and worse yet, if you read the detailed log messages (by setting GD_LEVEL to DUMP for instance) it will report “failed with error 126: The specified module could not be found” although the file *is* there.

So first try to understand what are the missing dependencies. Under Windows, use DependencyWalker (see <http://dependencywalker.com>) to check that a given DLL finds all its dependencies. If some dependencies are not found either:

- change your PATH so that it goes via the directories that contain the dependencies of your DLLs;
- or copy these dependencies in the ‘bin/’ directory of Urbi SDK, since that’s the directory of the program, ‘urbi-launch’.

The first approach is more tractable. Beware that dependencies may also have dependencies...

Cygwin Use the `cygcheck.exe` program to check dependencies. Beware that you must provide a qualified path to the file. Chances are that if

Shell Session

```
$ cygcheck foo.dll
```

does not work and will pretend that ‘`foo.dll`’ does not exist (although it’s *right there*), then this will work:

Shell Session

```
$ cygcheck ./foo.dll
```

In this output, look for lines like these:

Shell Session

```
cygcheck: track_down: could not find OgreMain.dll
cygcheck: track_down: could not find OIS.dll
cygcheck: track_down: could not find libuobject-vc90.dll
```

and make sure that ‘`OgreMain.dll`’, ‘`OIS.dll`’ and so forth are visible in the PATH (don’t be worry about ‘`libuobject-vc90.dll`’, `urbi-launch` will take care of it). Note that when they are finally visible from the PATH, then you can run

Shell Session

```
$ cygcheck OgreMain.dll
```

without having to specify the path.

18.3 urbiscript

18.3.1 Objects lifetime

18.3.1.1 How do I create a new Object derivative?

Urbi is based on prototypes. To create a new Object derivative (which will inherit all the Object methods), you can do:

urbiscript Session

```
var myObject = Object.new;
[00000001] Object_0x76543210
```

18.3.1.2 How do I destroy an Object?

There is no `delete` in Urbi, for a number of reasons (see ??). Objects are deleted when they are no longer used/referenced to.

In practice, users who want to “delete an object” actually want to remove a slot — see [Section 9.1](#). Users who want to clear an object can empty it — see ??.

Note that `myObject = nil` does not explicitly destroy the object bound to the name `myObject`, yet it may do so provided that `myObject` was the last and only reference to this object.

18.3.2 Slots and variables

18.3.2.1 Is the lobby a scope?

One frequently asked question is what visibility do variables have in urbiscript, especially when they are declared at the top-level interactive loop. In this section, we will see the mechanisms behind slots, local variables and scoping to fully explain this behavior and determine how to proceed to give the right visibility to variables.

For instance, this code might seem confusing at first:

```
var mind = 42;
[00000002] 42
function get()
{
    echo(mind);
}|;
get();
[00000003] *** 42
function Object.get()
{
    echo(mind)
}|;
// Where is my mind?
Object.get;
[00000004:error] !!! lookup failed: mind
[00000004:error] !!!      called from: get
```

urbiscript
Session

Local variables, slots and targets The first point is to understand the difference between local variables and slots. Slots are simply object fields: a name in an object referring to another object, like members in C++. They can be defined with the `setSlot` method, or with the `var` keyword.

```
// Add an 'x' slot in Object, with value 51.
Object.setSlot("x", 51);
[00000000] 51
// This is an equivalent version, for the 'y' slot.
var Object.y = 51;
[00000000] 51

// We can access these slots with the dot operator.
Object.x + Object.y;
[00000000] 102
```

urbiscript
Session

On the other hand, local variables are not stored in an object, but in the execution stack: their lifetime spans from their declaration point to the end of the current scope. They are declared with the ‘`var`’ keyword.

```
function foo()
{
    // Declare an 'x' local variable, with value 51.
    var x = 51;
    // 'x' isn't stored in any object. It's simply
    // available until the end of the scope.
    echo(x);
};;
```

urbiscript
Session

You probably noticed that in the last two code snippets, we used the `var` keyword to declare both a slot in `Object` and a local variable. The rule is simple: `var` declares a slot if an owning object is specified with the dot notation, as in `var owner.slot`, and a local variable if only an unqualified name is given, as in `var name`.

```
{
```

urbiscript
Session

```
// Store a 'kyle' slot in Object.
var Object.kyle = 42;
// Declare a local variable, limited to this scope.
var kenny = 42;
}; // End of scope.
[00000000] 42

// Kyle survived.
echo(Object.kyle);
[00000000] *** 42

// Oh my God, they killed Kenny.
echo(kenny);
[00000000:error] !!! lookup failed: kenny
```

There is however an exception to this rule: `do` and `class` scopes are designed to define a target where to store slots. Thus, in `do` and `class` scopes, even unqualified `var` uses declare slots in the target.

urbiscript
Session

```
// Classical scope.
{
  var arm = 64; // Local to the scope.
};
[00000000] 64

// Do scope, with target Object
do (Object)
{
  var chocolate = 64; // Stored as a slot in Object.
};
[00000000] Object

// No arm...
echo(arm);
[00000000:error] !!! lookup failed: arm
// ... but still chocolate!
echo(chocolate);
[00000000] *** 64
```

Last tricky rule you must keep in mind: the top level of your connection — your interactive session — is a `do` (`lobby`) scope. That is, when you type `var x` directly in your connection, it stores an `x` slot in the `lobby` object. So, what is this `lobby`? It's precisely the object designed to store your top-level variables. Every Urbi server has an unique `Lobby` (note the capital), and every connection has its `lobby` that inherits the `Lobby`. Thus, variables stored in `Lobby` are accessible from any connection, while variables stored in a connection's `lobby` are local to this connection.

To fully understand how lobbies and the top-level work, we must understand how calls — message passing — work in urbiscript. In urbiscript, every call has a target. For instance, in `Object.x`, `Object` is the target of the `x` call. If no target is specified, as in `x` alone, the target defaults to `this`, yielding `this.x`. Knowing this rules, plus the fact that at the top-level `this` is `lobby`, we can understand better what happens when defining and accessing variables at the top-level:

urbiscript
Session

```
// Since we are at the top-level, this stores x in the lobby.
// It is equivalent to 'var lobby.x'.
var x = "hello";
[00000000] "hello"

// This is an unqualified call, and is thus
// equivalent to 'this.x'.
// That is, 'lobby.x' would be equivalent.
x;
```

```
[00000000] "hello"
```

Solving the tricky example We now know all the scoping rules required to explain the behavior of the first code snippet. First, let's determine why the first access to `mind` works:

```
// This is equivalent to 'var lobby.myMind = 42'.
var myMind = 42;
[00000001] 42
// This is equivalent to 'function lobby.getMine...'
function getMine()
{
    // This is equivalent to 'echo(this.myMind)'
    echo(myMind);
}
// This is equivalent to 'this.getMine()', i.e. 'lobby.getMine()' .
getMine();
[00000000] *** 42
```

urbiscript
Session

Step by step:

- We create a `myMind` slot in `lobby`, with value 42.
- We create a `getMine` function in `lobby`.
- We call the `lobby`'s `getMine` method.
- We access `this.myMind` from within the method. Since the method was called with `lobby` as targetMine, `this` is `lobby`, and `lobby.x` resolves to the previously defined 42.

We can also explain why the second test fails:

```
// Create the 'hisMind' slot in the lobby.
var hisMind = 42;
[00000000] 42
// Define a 'getHis' method in 'Object'.
function Object.getHis()
{
    // Equivalent to echo(this.hisMind).
    echo(hisMind)
}
// Call Object's getHis method.
Object.getHis;
[00000000:error] !!! lookup failed: hisMind
[00000000:error] !!!      called from: getHis
```

urbiscript
Session

Step by step:

- We create a `hisMind` slot in `lobby`, with value 42, like before.
- We create a `getHis` function in `Object`.
- We call `Object`'s `getHis` method.

In the method, `this` is `Object`. Thus `hisMind`, which is `this.hisMind`, fails because `Object` has no such slot.

The key to understanding this behavior is that any unqualified call — unless it refers to a local variable — is destined to `this`. Thus, variables stored in the lobby are only accessible from the top-level, or from functions that are targeted on the lobby.

So, where to store global variables? From these rules, we can deduce a simple statement: since unqualified slots are searched in `this`, for a slot to be global, it must always be accessible through `this`. One way to achieve this is to store the slot in `Object`, the ancestor of any object:

urbiscript
Session

```
var Object.global = 1664;
[00000000] 1664

function any_object()
{
    // This is equivalent to echo(this.global)
    echo(global);
}!;
```

In the previous example, typing `global` will look for the `global` slot in `this`. Since `this` necessarily inherits `Object`, it will necessarily be found.

This solution would work; however, storing all global variables in `Object` wouldn't be very clean. `Object` is rather designed to hold methods shared by all objects. Instead, a `Global` object exists. This object is a prototype of `Object`, so all his slots are accessible from `Object`, and thus from anywhere. So, creating a genuine global variable is as simple as storing it in `Global`:

urbiscript
Session

```
var Global.g = "I'm global!";
[00000000] "I'm global!"
```

Note that you might want to reproduce the `Global` system and create your own object to store your related variables in a more tidy fashion. This is for instance what is done for mathematical constants:

urbiscript
Session

```
// Store all constants here
class Constants
{
    var Pi = 3.14;
    var Euler = 2.17;
    var One = 1;
    // ...
}!;

// Make them global by making them accessible from Global.
Global.addProto(Constants);
[00000000] Global

// Test it.
Global.Pi;
[00000000] 3.14
Pi;
[00000000] 3.14
function Object.testPi() { echo(Pi) }!;
42.testPi;
[00000000] *** 3.14
```

18.3.2.2 How do I add a new slot in an object?

To add a slot to an object `O`, you have to use the `var` keyword, which is syntactic sugar for the `setSlot` method:

urbiscript
Session

```
var O2 = Object.new |
// Syntax...
var O2.mySlot1 = 42;
[00000001] 42

// and semantics.
O2.setSlot("mySlot2", 23);
[00000001] 23
```

Note that in a method, `this` designates the current object. It is needed to distinguish the name of a slot in the current object, versus a local variable name:

```
{
    // Create a new slot in the current object.
    var this.bar = 42;

    // Create a local variable, which will not be known anymore
    // after we exit the current scope.
    var qux = 23;
}

qux;
[00000001:error] !!! lookup failed: qux
bar;
[00000001] 42
```

urbiscript
Session

18.3.2.3 How do I modify a slot of my object?

Use the `=` operator, which is syntactic sugar for the `Object.updateSlot` method.

```
class O
{
    var mySlot = 42;
}
// Sugarful.
O.mySlot = 51;
[00000001] 51

// Sugar-free.
O.updateSlot("mySlot", 23);
[00000001] 23
```

urbiscript
Session

18.3.2.4 How do I create or modify a local variable?

Use `var` and `=`.

```
// In two steps: definition, and initial assignment.
var myLocalVariable;
myLocalVariable = "foo";
[00000001] "foo"
// In a single step: definition with an initial value.
var myOtherLocalVariable = "bar";
[00000001] "bar"
```

urbiscript
Session

18.3.2.5 How do I make a constructor?

You can define a method called `init` which will be called automatically by `new`. For example:

```
class myObject
{
    function init(x, y)
    {
        var this.x = x;
        var this.y = y;
    };
}
myInstance = myObject.new(10, 20);
```

urbiscript
Session

18.3.2.6 How can I manipulate the list of prototypes of my objects?

The `protos` method returns a list (which can be manipulated) containing the list of your object prototype.

urbiscript
Session

```
var myObject = Object.new;
myObject.protos;
[00000001] [Object]
```

18.3.2.7 How can I know the slots available for a given object?

The `Object.localSlotNames` and `Object.allSlotNames` methods return respectively the local slot names and the local+inherited slot names.

18.3.2.8 How do I create a new function?

Functions are first class objects. That means that you can add them as any other slot in an object:

urbiscript
Session

```
var myObject = Object.new;
var myObject.myFunction = function (x, y)
{ echo ("myFunction called with " + x + " and " + y );};
```

You can also use the following notation to add a function to your object:

urbiscript
Session

```
var myObject = Object.new;
function myObject.myFunction (x, y) { /* ... */};
```

or even group definitions within a `do` scope, which will automatically define new slots instead of local variables and functions:

urbiscript
Session

```
var myObject = Object.new;
do (myObject)
{
  function myFunction (x, y) { /* ... */};
};
```

or group those two statements by using a convenient `class` scope:

urbiscript
Session

```
class myObject
{
  function myFunction (x, y) { /* ... */};
};
```

18.3.3 Tags

See [Section 14.3](#), in the urbiscript User Manual, for an introduction about Tags. Then for a definition of the Tag objects (construction, use, slots, etc.), see [Tag](#).

18.3.3.1 How do I create a tag?

See [Section 24.66.2](#).

18.3.3.2 How do I stop a tag?

Use the `stop` method (see [Section 24.66.1.1](#)).

urbiscript
Session

```
myTag.stop;
```

18.3.3.3 Can tagged statements return a value?

Yes, by giving it as a parameter to `stop`. See [Section 24.66.1.1](#).

18.3.4 Events

See [Chapter 15](#), in the urbiscript User Manual, for an introduction about event-based programming. Then for a definition of the Event objects (construction, use, slots, etc.), see [Event](#).

18.3.4.1 How do I create an event?

Events are objects, and must be created as any object by using `new` to create derivatives of the `Event` object.

```
var ev = Event.new;
```

urbiscript
Session

See [Section 24.15.3](#).

18.3.4.2 How do I emit an event?

Use the `!` operator.

```
ev!(1, "foo");
```

urbiscript
Session

18.3.4.3 How do I catch an event?

Use the `at(event?args)` construct (see [Section 23.10.1](#)).

```
at(ev?(1, var msg))
  echo ("Received event with 1 and message " + msg);
```

urbiscript
Session

The `?` marker indicates that we are looking for an event instead of a Boolean condition. The construct `var msg` indicates that the `msg` variable will be bound (as a local variable) in the body part of the `at` construct, with whatever value is present in the event that triggered the `at`.

18.3.5 Standard Library

18.3.5.1 How can I iterate over a list?

Use the `for` construct `(??)`, or the `List.each` method:

```
for (var i: [10, 11, 12]) echo (i);
[00000001] *** 10
[00000002] *** 11
[00000003] *** 12
```

urbiscript
Session

18.4 UObjects

18.4.1 Is the UObject API Thread-Safe?

We are receiving a lot of questions on thread-safety issues in UObject code. So here comes a quick explanation on how things work in plugin and remote mode, with a focus on those questions.

18.4.1.1 Plugin mode

In *plugin mode*, all the UObject callbacks (timer, bound functions, notifyChange and notifyAccess targets) are called synchronously in the same thread that executes urbscript code. All reads and writes to Urbi variables, through *UVar*, are done synchronously. Access to the UObject API (reading/writing UVars, using `call()`...) is possible from other threads, though those operations are currently using one serialization lock with the main thread: each UObject API call from another thread will wait until the main thread is ready to process it.

18.4.1.2 Remote mode

Execution model In *remote mode*, a single thread is also used to handle all UObject callbacks, for all the UObject in the same executable. It means that two bound functions registered from the same executable will never execute in parallel. Consider this sample C++ function:

```
C++ int MyObject::test(int delay)
{
    static const int callNumber = 0;
    int call = ++callNumber;
    std::cerr << "in " << call << ":" << time() << std::endl;
    sleep(delay);
    std::cerr << "out " << call << ":" << time() << std::endl;
    return 0;
}
```

If this function is bound in a remote uobject, the following code:

```
C++ MyObject.test(1), MyObject.test(1)
```

will produce the following output (assuming the first call to `time` returns 1000).

```
in 1: 1000
out 1: 1001
in 2: 1001
out 2: 1002
```

However, the execution of the Urbi kernel is not “stuck” while the remote function executes, as the following code demonstrates:

urbscript
Session

```
var t = Tag.new;
test(1) | t.stop,
t:every(300ms)
    cerr << "running";
```

The corresponding output is (mixing the kernel and the remote outputs):

```
[0] running
in 1: 1000
[300] running
[600] running
[900] running
out 1: 1001
```

As you can see, Urbi semantics is respected (the execution flow is stuck until the return value from the function is returned), but the kernel is not stuck: other pieces of code are still running.

Thread-safety The liburbi and the UObject API in remote mode are thread safe. All operations can be performed in any thread. As always, care must be taken for all non-atomic operations. For example, the following function is not thread safe:

```
C++ void writeToVar(UClient* cl, std::string varName, std::string value)
{
```

```

    (*cl) << varName << " = " << value << ";" ;
}
```

Two simultaneous calls to this function from different threads can result in the two messages being mixed. The following implementation of the same function is thread-safe however:

```

void
writeToVar(UClient* cl, std::string varName, std::string value)
{
    std::stringstream s;
    s << varName << " = " << value << ";";
    (*cl) << s.str();
}
```

C++

since a single call to UClient's `operator <<` is thread-safe.

18.5 Miscellaneous

18.5.1 What has changed since the latest release?

See [Chapter 36](#).

18.5.2 How can I contribute to the code?

You are encouraged to submit patches to kernel@lists.gostai.com, where they will be reviewed by the Urbi team. If they fit the project and satisfy the quality requirements, they will be accepted. As of today there is no public repository for Urbi SDK (there will be, eventually), patches should be made against the latest source tarballs (see <http://www.gostai.com/downloads/urbi/2.x/>).

Even though Urbi SDK is free software (GNU Affero General Public License 3+, see the 'LICENSE.txt' file), licensing patches under GNU AGPL3+ does not suffice to support our dual licensed products. This situation is common, see for instance the case of Oracle VM Virtual Box, http://www.virtualbox.org/wiki/Contributor_information.

There are different means to ensure that your contributions to Urbi SDK can be accepted. None require that you “give away your copyright”. What is needed, is the right to use contributions, which can be achieved in two ways:

- Sign the Urbi Open Source Contributor Agreement, see [Section 37.10](#). This may take some time initially, but it will cover all your future contributions.
- Submit your contribution under the Expat license (also known as the “MIT license”, see [Section 37.3](#)), or under the modified BSD license (see [Section 37.2](#)).

18.5.3 How do I report a bug?

Bug reports should be sent to kernel-bugs@lists.gostai.com, it will be addressed as fast as possible. Please, be sure to read the FAQ (possibly updated on our web site), and to have checked that no more recent release fixed your issue.

Each bug report should contain a self-contained example, which can be tested by our team. Using self-contained code, i.e., code that does not depend on other code, helps ensuring that we will be able to duplicate the problem and analyze it promptly. It will also help us integrating the code snippet into our non-regression test suite so that the bug does not reappear in the future.

If your report identifies a bug in the Urbi kernel or its dependencies, we will prepare a fix to be integrated in a later release. If the bug takes some time to fix, we may provide you with a workaround so that your developments are not delayed.

In your bug report, specify the Urbi version you are using (run ‘`urbi --version`’) and whether this bug is blocking you or not. Please keep kernel-bugs@lists.gostai.com in copy

of all your correspondence: do not reply individually to a member of our team, as this may slow down the handling of the report.

If your bug report is about a failing ‘`make check`’, first be sure to read [Section 21.9](#).

Chapter 19

Urbi Guideline

19.1 urbiscript Programming Guideline

19.1.1 Prefer Expressions to Statements

Code like this:

```
if (delta)
    cmd = "go";
else if (alpha)
    cmd = "turn"
else
    cmd = "stop";
```

urbiscript
Session

is more legible as follows:

```
cmd =
{
    if      (delta) "go"
    else if (alpha) "turn"
    else          "stop"
};
```

urbiscript
Session

19.1.2 Avoid `return`

The `return` statement is actually costly, because it also in charge of stopping all the event-handlers, detached code, and code sent into background (via ‘,’ or ‘&’).

In a large number of cases, `return` is actually used uselessly. For instance instead of:

```
function inBounds1(var x, var low, var high)
{
    if (x < low)
        return false;
    if (high < x)
        return false;
    return true;
}|;
assert
{
    inBounds1(1, 0, 2);    inBounds1(0, 0, 2);  inBounds1(2, 0, 2);
    !inBounds1(0, 1, 2);  !inBounds1(3, 0, 2);
};
```

urbiscript
Session

write

```
function inBounds2(var x, var low, var high)
{
    if (x < low)
        false
```

urbiscript
Session

```

    else if (high < x)
        false
    else
        true;
}|;
assert
{
    inBounds2(1, 0, 2);    inBounds2(0, 0, 2);  inBounds2(2, 0, 2);
    !inBounds2(0, 1, 2);  !inBounds2(3, 0, 2);
};

```

or better yet, simply evaluate the Boolean expression. As a matter of fact, returning a Boolean is often the sign that you ought to “return” a Boolean expression rather than evaluate it in a conditional, and return either true or false.

urbiscript
Session

```

function inBounds3(var x, var low, var high)
{
    low <= x && x <= high
}|;
assert
{
    inBounds3(1, 0, 2);    inBounds3(0, 0, 2);  inBounds3(2, 0, 2);
    !inBounds3(0, 1, 2);  !inBounds3(3, 0, 2);
};

```

Chapter 20

Migration from urbiscript 1 to urbiscript 2

This chapter is intended to people who want to migrate programs in urbiscript 1 to urbiscript 2. Backward compatibility is *mostly* ensured, but some urbiscript 1 constructs were removed because they prevented the introduction of cleaner constructs in urbiscript 2. When possible, urbiscript 2 supports the remaining urbiscript 1 constructs. The [Kernel1](#) object contains functions that support some urbiscript 1 features.

20.1 \$(Foo)

This construct was designed to build identifiers at run-time. This used to be a common idiom to work around some limitations of urbiscript 1 which are typically *no longer needed in urbiscript 2*. For instance, genuine local variables are simpler and safer to use than identifiers forged by hand to be unique. In order to associate information to a string, use a [Dictionary](#).

If you really need to forge identifiers at run-time, use `setSlot`, `updateSlot`, and `getSlot`, which all work with strings, and possibly `asString`, which converts arbitrary expressions into strings. The following table lists common patterns.

Deprecated	Updated
<code>var \$(Foo) = ...;</code> <code>\$(Foo) = ...;</code> <code>\$(Foo)</code>	<code>setSlot(Foo.asString, ...);</code> <code>updateSlot(Foo.asString, ...);</code> <code>getSlot(Foo.asString);</code>

20.2 delete Foo

In order to maintain an analogy with the C++ language, urbiscript used to support `delete Foo`, but this was removed for a number of reasons:

- urbiscript 2 features genuine local variables for which `delete` makes no sense.
- in C++ `delete` really targets the object: destroy yourself, then the system will reclaim the memory. In urbiscript one cannot destroy an object and reclaim the memory, it is the task of the system to notice objects that are no longer used, and to reclaim the memory. This is called *garbage collection*. Therefore in urbiscript `delete` is actually bounced to `Object.removeLocalSlot` sent to the owner of the object.
- `delete` is an unsafe feature that makes only sense in pointer-based languages such as C and C++. It enables nice bugs such as:

```
var this.a := A.new;
// ...
```

urbiscript
Session

```
delete this.a;
// ...
cout << this.a;
```

For these reasons, and others, `delete` Foo was removed. To remove the *name* Foo, run `removeLocalSlot("Foo")` ([Section 9.1](#)) — the garbage collector will reclaim memory if there are no other use of Foo. To remove the contents of Foo, you remove all its slots one by one:

urbiscript
Session

```
class Foo
{
    var a = 12;
    var b = 23;
} | {};
function Object.removeAllSlots()
{
    for (var s: localSlotNames)
        removeLocalSlot(s);
} | {};
Foo.removeAllSlots;
Foo.localSlotNames;
[00000000] []
```

20.3 emit Foo

The keyword `emit` is deprecated in favor of `!`.

Deprecated	Updated
<code>emit e;</code>	<code>e!;</code>
<code>emit e(a);</code>	<code>e!(a);</code>
<code>emit e ~ 1s;</code>	<code>e! ~ 1s;</code>
<code>emit e(a) ~ 1s;</code>	<code>e!(a) ~ 1s;</code>

The `?` construct is changed for symmetry.

Deprecated	Updated
<code>at (?e)</code>	<code>at (e?)</code>
<code>at (?e(var a))</code>	<code>at (e?(var a))</code>
<code>at (?e(var a) if a == 2)</code>	<code>at (e?(var a) if a == 2)</code>

20.4 eval(Foo)

`eval` is still supported, but its use is discouraged: one can often easily do without. For instance, `eval` was often used to manipulate forged identifiers; see [Section 20.1](#) for means of getting rid of them.

20.5 foreach

The same feature with a slightly different syntax is now provided by `for`. See [Section 23.7.5.2](#).

20.6 group

Where support for groups was a built-in feature in urbiscript 1, it is now provided by the standard library, see [Group](#). Instead of

urbiscript
Session

```
group myGroup {a, b, c}
```

write

```
var myGroup = Group.new(a,b,c)
```

urbiscript
Session

20.7 loopn

The same feature and syntax is now provided by [for](#). See [Section 23.7.5.3](#).

20.8 new Foo

See [Section 12.5](#) for details on `new`. The construct `new Foo` is no longer supported because it is (too) ambiguous: what does `new a(1,2).b(3,4)` mean? Is `a(1,2).b` the object to clone and `(3,4)` are the arguments of the constructor? Or is it the result of `a(1,2).b(3,4)` that must be cloned?

In temporary versions, urbiscript 2 used to support this `new` construct, but too many users got it wrong, and we decided to keep only the simpler, safer, and more consistent method-call-style construct: `Foo.new`. Every single possible interpretation of `new a(1,2).b(3,4)` is reported below, unambiguously.

- `a(1,2).b(3,4).new`
- `a(1,2).b.new(3,4)`
- `a(1,2).new.b(3,4)`
- `a.new(1,2).b(3,4)`
- `new.a(1,2).b(3,4)`

20.9 self

For consistency with the C++ syntax, urbiscript now uses `this`.

20.10 stop Foo

Use `Foo.stop` instead, see [Tag](#).

20.11 # line

Use `//#line` instead, see [Section 23.1.3](#).

20.12 tag+end

To detect the end of a statement, instead of

```
mytag+end: { echo ("foo") },
```

urbiscript
Session

use the `leave?` method of the [Tag](#) object:

```
{
    var mytag = Tag.new("mytag");
    at (mytag.leave?)
        Channel.new("mytag") << "code has finished";
    mytag: { echo ("foo") },
};

[00000002] *** foo
[00000003:mytag] "code has finished"
```

urbiscript
Session

Chapter 21

Building Urbi SDK

This section is meant for people who want to *build* the Urbi SDK. If you just want to install a pre-built Urbi SDK, see [Chapter 17](#).

A foreword that applies to any package, not just Urbi SDK: **building or checking as root is a bad idea**. Build as a regular user, and run ‘`sudo make install`’ just for the install time *if you need privileges to install to the chosen destination*.

21.1 Requirements

This section details the dependencies of this package. Some of them are required to *bootstrap* the package ([Section 21.1.1](#)), i.e., to build it from the repository. Others are required to *build* the package ([Section 21.1.2](#)), i.e., to compile it from a tarball, or after a bootstrap. Finally, some are needed to run the test suite ([Section 21.1.3](#)).

The reader in a hurry can simply run one of the following commands, depending on her environment.

Debian or Ubuntu

```
sudo apt-get install \
    aspell aspell-en autoconf automake bc ccache colordiff \
    coreutils cvs doxygen flex g++ gettext git-core gnuplot \
    graphviz help2man imagemagick libboost-all-dev \
    liblinphone-dev libx11-dev make pkg-config \
    python-docutils python-yaml python2.6 socat \
    sun-java6-jdk swig tex4ht texinfo texlive-base \
    texlive-binaries texlive-latex-extra transfig valgrind
```

Shell
Session

MacPorts

```
sudo port install \
    ImageMagick aspell aspell-dict-en autoconf automake bc \
    boost ccache colordiff coreutils cvs doxygen flex gcc42 \
    gettext git-core gmake gnuplot graphviz help2man libxslt \
    linphone pkgconfig py26-docutils py26-yaml python26 \
    python_select socat swig swig-java texinfo texlive \
    texlive-bin-extra texlive-htmlxml texlive-latex-extra \
    transfig valgrind-devel
```

Shell
Session

21.1.1 Bootstrap

To bootstrap this package from its repository, and then to compile it, the following tools are needed.

Autoconf 2.64 or later

```
package: autoconf
```

Automake 1.11.1 or later Note that if you have to install Automake by hand (as opposed to “with your distribution’s system”), you have to tell its `aclocal` that it should also look at the files from the system’s `aclocal`. If `/usr/local/bin/aclocal` is the one you just installed, and `/usr/bin/aclocal` is the system’s one, then run something like this:

Shell Session

```
$ dirlist=$(./usr/local/bin/aclocal --print-ac-dir)/dirlist
$ sudo mkdir -p $(dirname $dirlist)
$ sudo /usr/bin/aclocal --print-ac-dir >>$dirlist
```

```
package: automake
```

Cvs This surprising requirement comes from the system Bison uses to fetch the current version of the message translations.

```
package: cvs
```

colordiff Not a requirement, but a useful addition. Used if exists.

```
package: colordiff
```

Git 1.6 or later Beware that older versions behave poorly with submodules.

```
package: git-core
```

Gettext 1.17 Required to bootstrap Bison. In particular it provides autopoint.

```
package: gettext
```

GNU sha1sum We need the GNU version of sha1sum.

```
package: coreutils
```

Help2man Needed by Bison.

```
package: help2man
```

Python You need ‘`xml/sax`’, which seems to be part only of Python 2.6. Using `python_select` can be useful to state that you want to use Python 2.6 by default (`sudo python_select python26`).

```
deb: python2.6
MacPorts: python26 python_select
```

Texinfo Needed to compile Bison.

```
package: texinfo
```

yaml for Python The AST is generated from a description written in YAML. Our (Python) tools need to read these files to generate the AST classes. See <http://pyyaml.org/wiki/PyYAML>. The installation procedure on Cygwin is:

Shell Session

```
$ cd /tmp
$ wget http://pyyaml.org/download/pyyaml/PyYAML-3.09.zip
$ unzip PyYAML-3.09.zip
$ cd pyYAML-3.09
$ python setup.py install
```

```
MacPorts: py26-yaml
deb: python-yaml
```

On OS X you may also need to specify the PYTHONPATH:

```
export PYTHONPATH="\n
/opt/local/Library/Frameworks/Python.framework/Versions/2.6\n
/lib/python2.6/site-packages:$PYTHONPATH"
```

Shell Session

21.1.2 Build

Boost 1.40 or later Don't try to build it yourself, ask your distribution's management to install it.

```
deb: libboost-all-dev
MacPorts: boost
windows: http://www.boostpro.com/download
```

Ccache Not a requirement, but it's better to have it.

```
package: ccache
```

Doxxygen Needed if you enable the Doxygen documentation (via `configure`'s option '`--enable-documentation=doxygen,...`).

```
package: doxygen
```

Flex 2.5.35 or later Beware that 2.5.33 has bugs than prevents this package from building correctly.

```
package: flex
```

G++ 4.0 or later GCC 4.2 or later is a better option. You may have a problem with GCC 4.4 which rejects some Bison generated code. See [Section 18.1.2](#).

```
deb: g++
MacPorts: gcc42
```

Gnuplot Required to generate charts of trajectories for the documentation.

```
package: gnuplot
```

GraphViz Used to generate some of the figures in the documentation. There is no GraphViz package for Cygwin, so download the MSI file from GraphViz' site, and install it. Then change your path to go into its bin directory.

```
wget http://www.graphviz.org/pub/graphviz/stable/windows/graphviz-2.26.msi
PATH=/cygdrive/c/Program\ Files/Graphviz2.26/bin:$PATH
```

Shell Session

```
package: graphviz
```

ImageMagick Used to convert some of the figures in the documentation.

```
MacPorts: ImageMagick
deb: imagemagick
```

JDK In order to compile support for Java UObjects, you need Sun's JDK. In particular the ‘jni.h’ must be available. On OS X Snow Leopard, you need the “Java Developer” package from Apple. See <http://connect.apple.com/>.

```
deb: sun-java6-jdk
```

GNU Make Although we use Automake that does provide portability across flavors of Make, we do use GNU Make extensions. Actually, the most common version, 3.81, behaves improperly on our Makefiles, so be sure to use GNU Make 3.82.

```
deb: make
MacPorts: gmake
```

oRTP The UObject middleware can use the RTP protocol to provide better throughput for streams. The implementation relies on the oRTP library which is now part of the linphone package. oRTP is not needed, but it is strongly recommended.

```
MacPorts: linphone
deb: liblinphone-dev
```

PDFLaTeX TeX Live is the most common TeX distribution nowadays. We use pdfcrop from ‘texlive-bin-extra’.

```
deb: texlive-base texlive-latex-extra texlive-binaries
MacPorts: texlive texlive-latex-extra texlive-bin-extra
```

pkg-config To find some libraries (such as oRTP), we use `pkg-config`.

```
deb: pkg-config
MacPorts: pkgconfig
```

py-docutils This is not a requirement, but it’s better to have it. Used by the test suite. Unfortunately the name of the package varies between distributions. It provides `rst2html`.

```
deb: python-docutils
MacPorts: py26-docutils
```

ROS CTurtle or later Needed to enable our ROS bridge. Versions CTurtle, Diamondback, and Electric Emys (1.2.X, 1.4.X, 1.6.X) are known to work. The packages and installation processes are detailed at <http://www.ros.org/wiki/ROS>.

SWIG 1.3.36 or later Used to generate the Java support for UObject. Versions 1.3.36, 1.3.40, and 2.0.0 are known to work.

```
package: swig
MacPorts: swig-java
```

TeX4HT Used to generate the HTML documentation.

```
deb: tex4ht
MacPorts: texlive-htmlxml
```

Transfig Needed to convert some figures for documentation (using fig2dev).

```
package: transfig
```

x11 Client-side library, used to generate `urbi-image`.

```
deb: libx11-dev
```

xsltproc This is not a requirement, but it's better to have it. Used to generate reports about the urbscript grammar.

```
MacPorts: libxslt
```

21.1.3 Check

aspell Needed by the test suite.

```
deb: aspell aspell-en
MacPorts: aspell aspell-dict-en
```

bc Needed by the test suite.

```
package: bc
```

socat Needed by the test suite to send messages to an Urbi server.

```
package: socat
```

Valgrind Not needed, but if present, used by the test suite.

```
deb: valgrind
MacPorts: valgrind-devel
```

21.2 Check out

Get the open source version tarball from <http://www.gostai.com/downloads/urbi/2.x/> and uncompress it. This version is bootstrapped, you can directly proceed to the “Configure” step.

21.3 Bootstrap

Run

```
$ ./bootstrap
```

Shell
Session

21.4 Configure

Do not compile where the source files are (the test suite of a sub-component of ours, Libport, is known to fail in that situation). So, **compile in another directory than the one containing the sources**, for instance as follows.

```
$ mkdir _build
$ cd _build
$ ../configure options...
```

Shell
Session

Also, do not compile in a directory whose name, or the name of some of its ancestor, use “special characters”: stick to plain ASCII. This is a limitation of the Java VM which seems to be unable to traverse properly directories with UTF-8 names when the locale is set to C ([Section 18.1.4](#)).

The directory containing ‘`configure`’ is called the *source directory*, or *srcdir*, and the directory in which the compilation takes place (‘`_build`’ in this case) is named the *build directory*, or *builddir* for short. Finally, the root in which everything will be installed is called by several names: `URBI_ROOT`, the *prefix*, or the *install dir*, aka *installdir*.

21.4.1 configuration options

See ‘`../configure --help`’ for detailed information. Unless you want to do funky stuff, you probably need no option.

To use `ccache`, pass ‘`CC='ccache gcc' CXX='ccache g++'`’ as arguments to `configure`:

Shell Session

```
$ ../configure CC='ccache gcc' CXX='ccache g++' ...
```

For the records, here are some of the ‘`--enable-`’/‘`--disable-`’ specific options.

‘`--disable-bindings`’

Whether to disable UObject Java support.

‘`--enable-compilation-mode=mode`’

Specify the flavor of the build. You should use ‘`--enable-compilation-mode=speed`’.

‘`--enable-doc=formats`’

Specify what formats of the documentation must be built. Disabling is fine: ‘`--disable-doc`’.

‘`--enable-doc-sections=sections`’

If the PDF or HTML documentation is to be generated, specify what optional sections of the documentation must be built.

‘`--enable-examples`’

Whether to build the sample programs (`urbi-ball-tracking` and so forth). Disabling is fine: ‘`--disable-examples`’.

‘`--enable-library-suffix=suffix`’

Specify that libraries should be named ‘`libfoosuffix.*`’ instead of ‘`libfoo.*`’. If *suffix* is `auto` (or `yes`, or if no argument is passed), then use the same suffix as Boost (e.g., ‘`-vc80-d`’ for Visual 2005, debug libraries). If *suffix* is `autodebug`, then just append ‘`-d`’ for debug builds.

‘`--enable-sdk-remote`’

Whether to build the Urbi SDK Remote libraries. Disabling is fine: ‘`--disable-sdk-remote`’. Some of the components that are shared between Urbi and SDK Remote will be built anyway, so there will be compilations to perform in ‘`sdk-remote`’.

‘`--enable-ufloat=kind`’

Do not use this option, it is experimental and unreliable.

21.4.2 Windows: Cygwin

The builds for Windows use our wrappers. These wrappers use a database to store the dependencies (actually, to speed up their computation). We use Perl, and its DBI module. So be sort to have sqlite, and DBI.

Shell Session

```
$ perl -MCPAN -e 'install Bundle::DBI'
```

It might fail. The most important part is

```
$ perl -MCPAN -e 'install DBD::SQLite'
```

Shell
Session

It might suffice...

21.4.3 Building For Windows

We support two builds: using Wine on top of Unix, and using Cygwin on top of Windows.

Both builds use our wrappers around MSVC's tools (`cl.exe`, `dumpbin.exe`, `link.exe`). It is still unclear whether it was a good or a bad idea, but the wrappers use the same names as the tools themselves. To set up Libtool properly, you will need to pass the following as argument to configure:

```
AR=lib.exe
CC='ccache cl.exe'
CC_FOR_BUILD=gcc
CXX='ccache cl.exe'
LD=link.exe
DUMPBIN='dumpbin.exe'
RANLIB=:
host_alias=mingw32
--host=mingw32
```

Shell
Session

where: '`cl.exe`', '`dumpbin.exe`', '`lib.exe`', and '`link.exe`' are the wrappers.

Since we are cross-compiling, we also need to specify `CC_FOR_BUILD` so that `config.guess` can properly guess the type of the build machine.

21.4.4 Building for Windows using Cygwin

We use our wrappers, which is something that Libtool cannot know. So we need to tell it that we are on Windows with Cygwin, and pretend we use GCC, so we pretend we run mingw.

The following options have been used with success to compile Urbi SDK with Visual C++ 2005. Adjust to your own case (in particular the location of Boost and the '`vcxx8`' part).

```
$ ./configure
-C
--prefix=/usr/local/gostai
--enable-compilation-mode=speed
--enable-shared
--disable-static
--enable-dependency-tracking
--with-boost=/cygdrive/c/gt-win32-2/d/boost_1_40
AR=lib.exe
CC='ccache cl.exe'
CC_FOR_BUILD=gcc
CXX='ccache cl.exe'
LD=link.exe
DUMPBIN=dumpbin.exe
RANLIB=:
host_alias=mingw32
--host=mingw32
```

Shell
Session

21.5 Compile

Should be straightforward.

```
$ make -j3
```

Shell
Session

Using `distcc` and `ccache` is recommended.

21.6 Install

Running ‘`make install`’ works as expected. It is a good idea to check that your installation works properly: run ‘`make installcheck`’ (see [Section 21.9](#)).

21.7 Relocatable

If you intend to make a relocatable tree of Urbi SDK (i.e., a self-contained tree that can be moved around), then run ‘`make relocatable`’ after ‘`make install`’.

This step *requires* that you use a `DESTDIR` (see the Automake documentation for more information). Basically, the sequence of commands should look like:

Shell Session

```
$ DESTDIR=/tmp/install
$ rm -rf $DESTDIR
$ make install DESTDIR=$DESTDIR
$ make relocatable DESTDIR=$DESTDIR
$ make installcheck DESTDIR=$DESTDIR
```

You may now move this tree around and expect the executable to work properly.

21.8 Run

In addition to the “public” environment variables ([Section 22.1.2](#)), some other, reserved for developers, alter the behavior of the programs.

URBI_ACCEPT_BINARY_MISMATCH As a safety net, Urbi checks that loaded modules were compiled with exactly the same version of Urbi SDK. Define this variable to skip this check, at your own risks.

URBI_CHECK_MODE Skip lines in input that look like Urbi output. A way to accept test files (*.chk) as input.

URBI_DESUGAR Display the desugared ASTs instead of the original one.

URBI_DOC Where to find the filedoc directory, which contains ‘`THANKS.txt`’ and so forth.

URBI_IGNORE_URBI_U Ignore failures (such as differences between kernel revision and ‘`urbi.u`’ revision) during the initialization.

URBI_INTERACTIVE Force the interactive mode, as if ‘`--interactive`’ was passed.

URBI_LAUNCH The path to `urbi-launch` that `urbi.exe` will exec.

URBI_NO_ICE_CATCHER Don’t try to catch SEGVs.

URBI_PARSER Enable Bison parser traces.

URBI_REPORT Display statistics about execution rounds performed by the kernel.

URBI_ROOT_LIBname The location of the libraries to load, without the extension. The `LIBname` are: LIBJPEG4URBI, LIBPLUGIN (libuobject plugin), LIBPORT, LIBREMOTE (libuobject remote), LIBSCHED, LIBSERIALIZE, LIBURBI.

URBI_SCANNER Enable Flex scanner traces.

URBI_SHARE Where to find the fileshare directory, which contains ‘`images/gostai-logo`’, ‘`urbi/urbi.u`’ and so forth.

URBI_TEXT_MODE Forbid binary communications with UObjects.

`URBI_TOPOLEVEL` Force the display the result of the top-level evaluation into the lobby.

By carefully defining these variables, it is possible to run the non-installed programs. There are wrappers that do this properly:

```
urbi Use builddir/tests/bin/urbi.
umake builddir/sdk-remote/src/tests/bin/umake;
umake-java builddir/sdk-remote/src/tests/bin/umake-java;
umake-link builddir/sdk-remote/src/tests/bin/umake-link;
umake-shared builddir/sdk-remote/src/tests/bin/umake-shared;
urbi-launch builddir/sdk-remote/src/tests/bin/urbi-launch;
urbi-launch-java builddir/sdk-remote/src/tests/bin/urbi-launch-java;
urbi-send builddir/sdk-remote/src/tests/bin/urbi-send;
```

21.9 Check

Root Running the test-suite as a super-user (root) is a bad idea ([Chapter 21](#)): some tests check that Urbi SDK respects file permissions, which of course cannot work if you are omnipotent.

Parallel Tests There are several test suites that will be run if you run ‘`make check`’ (‘`-j4`’ works on most machines).

Some tests are extremely “touchy”. Because the test suite exercises Urbi under extreme conditions, some tests may fail not because of a problem in Urbi, but because of non-determinism in the test itself. In this case, another run of ‘`make check`’ will give an opportunity for the test to pass (remind that the tests that passed will not be run again). Also, using ‘`make check -j16`’ is a sure means to have the Urbi scheduler behave insufficiently well for the test to pass. **Do not send bug reports for such failures..** Before reporting bugs, make sure that the failures remain after a few ‘`make check -j1`’ invocations.

To rerun only the tests that failed, use ‘`make recheck`’. Some tests have explicit dependencies, and they are not rerun if nothing was changed (unless they had failed the previous time). It is therefore expected that after one (long) run of ‘`make check`’, the second one will be “instantaneous”: the previous log files are reused.

21.9.1 Lazy test suites

The test suites are declared as “lazy”, i.e., unless its dependencies changed, a successful test will be run only once — failing tests do not “cache” their results. Because spelling out the dependencies is painful, we rely on a few timestamps:

‘`libraries.stamp`’ Updated when a library is updated (libport, libuobject, etc.).

‘`executables.stamp`’ Updated when an executable is updated (`urbi-launch`, etc.). Depends on `libraries.stamp`.

‘`urbi.stamp`’ When Urbi sources (‘`share/urbi/*.u`’) are updated.

‘`all.stamp`’ Updated when any of the three aforementioned timestamps is.

These timestamps are updated **only** when `make` is run in the top-level. Therefore, the following sequence does not work as expected:

```
$ make check -C tests      # All passes.
$ emacs share/urbi/foo.u
$ make check -C tests      # All passes again.
```

because the timestamps were not given a chance to notice that some Urbi source changed, so Make did not notice the tests really needed to be rerun and **the tests were not run**.

You may either just update the timestamps:

Shell Session

```
$ make check -C tests      # All passes.
$ emacs share/urbi/foo.u
$ make stamps              # Update the timestamps.
$ make check -C tests      # All passes again.
```

or completely disable the test suite laziness:

Shell Session

```
$ make check -C tests LAZY_TEST_SUITE=
```

21.9.2 Partial test suite runs

You can run each test suite individually by hand as follows:

‘`sdk-remote/libport/test-suite.log`’ Tests libport.

Shell Session

```
$ make check -C sdk-remote/libport
```

‘`sdk-remote/src/tests/test-suite.log`’ Checks liburbi, and some of the executables we ship. Requires the kernel to be compiled in order to be able to test some of the uobjects.

Shell Session

```
$ make check -C sdk-remote/src/tests
```

‘`tests/test-suite.log`’ Tests the kernel and uobjects.

Shell Session

```
$ make check -C tests
```

Partial runs can be invoked:

Shell Session

```
$ make check -C tests TESTS='2.x/echo.chk'
```

wildcards are supported:

Shell Session

```
$ make check -C tests TESTS='2.x/*'
```

To check remote uobjects tests:

Shell Session

```
$ make check -C tests TESTS='uob/remote/*'
```

‘`doc/tests/test-suite.log`’ The snippets of code displayed in the documentation are transformed into test files.

Shell Session

```
$ make check -C doc
```

Partial runs for the doc tests:

Shell Session

```
$ make -C doc check-TESTS TESTS='tests/specs/float-00.chk'
```

Part IV

Urbi SDK Reference Manual

About This Part

This part defines the specifications of the urbiscript language. It defines the expected behavior from the urbiscript interpreter, the standard library, and the SDK. It can be used to check whether some code is valid, or browse urbiscript or C++ API for a desired feature. Random reading can also provide you with advanced knowledge or subtleties about some urbiscript aspects.

This part is not an urbiscript tutorial; it is not structured in a progressive manner and is too detailed. Think of it as a dictionary: one does not learn a foreign language by reading a dictionary. For an urbiscript Tutorial, see [Part II](#).

This part does not aim at giving advanced programming techniques. Its only goal is to define the language and its libraries.

[Chapter 22 — Programs](#)

Presentation and usage of the different tools available with the Urbi framework related to urbiscript, such as the Urbi server, the command line client, `umake`, ...

[Chapter 23 — urbiscript Language Reference Manual](#)

Core constructs of the language and their behavior.

[Chapter 24 — urbiscript Standard Library](#)

Listing of all classes and methods provided in the standard library.

[Chapter 25 — Communication with ROS](#)

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, see <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

[Chapter 26 — Gostai Standard Robotics API](#)

Also known as “The Urbi Naming Standard”: naming conventions in for standard hardware/software devices and components implemented as UObject and the corresponding slots/events to access them.

Chapter 22

Programs

22.1 Environment Variables

There is a number of environment variables that alter the behavior of the Urbi tools.

22.1.1 Search Path Variables

Some variables define *search-paths*, i.e., colon-separated lists of directories in which library files (urbiscript programs, UObjects and so forth) are looked for.

The tools have predefined values for these variables which are tailored for your installation — so that Urbi tools can be run without any special adjustment. In order to provide the user with a means to override or extend these built-in values, the path variables support a special syntax: a lone colon specifies where the standard search path must be inserted. See the following examples about URBI_PATH.

```
# Completely override the system path. First look for files in
# /home/jessie/urbi, then in /usr/local/urbi.
export URBI_PATH=/home/jessie/urbi:/usr/local/urbi

# Prepend the previous path to the default path. This is dangerous as
# it may result in some standard files being hidden.
export URBI_PATH=/home/jessie/urbi:/usr/local/urbi:

# First look in Jessie's directory, then the default location, and
# finally in /usr/local/urbi.
export URBI_PATH=/home/jessie/urbi::/usr/local/urbi

# Extend the default path, i.e., files that are not found in the
# default path will be looked for in Jessie's place, and then in
# /usr/local/urbi
export URBI_PATH=::/home/jessie/urbi:/usr/local/urbi
```

Shell
Session

Windows Issues

On Windows too directories are separated by colons, but backslashes are used instead of forward-slashes. For instance

```
URBI_PATH=C:\cygwin\home\jessie\urbi:C:\cygwin\usr\local\urbi
```

Shell
Session

22.1.2 Environment Variables

The following variables control the way the log messages are displayed. They are meant for developers.

GD_COLOR If set, force the use of colors in the logs, even if the output device does not seem to support them. See also **GD_NO_COLOR**.

GD_CATEGORY If set, a comma-separated list of categories or globs (e.g., ‘UValue’, ‘Urbi.*’) with modifiers ‘+’ or ‘-’. If the modifier is a ‘+’ or if there is no modifier the category will be displayed. If the modifier is a ‘-’ it will be hidden.

Modifiers can be chained, accumulated from left to right: ‘-*,+Urbi.*,-Urbi.At’ will only display categories beginning with ‘Urbi.’ except ‘Urbi.At’, ‘-Urbi.*’ will display all categories except those beginning with ‘Urbi.’.

Actually, the first character defines the state for unspecified categories: ‘Urbi’ (or ‘+Urbi’) enables only the ‘Urbi’ category, while ‘-Urbi’ enables everything but ‘Urbi’. Therefore, ‘+Urbi.*,-Urbi.At’ and ‘Urbi.*,-Urbi.At’ are equivalent to ‘-*,+Urbi.*,-Urbi.At’.

GD_ENABLE_CATEGORY If set, a comma-separated list of categories or globs (e.g., ‘UValue’, ‘Urbi.*’) to display. All the others will be discarded. See also **GD_DISABLE_CATEGORY**.

GD_DISABLE_CATEGORY If set, a comma-separated list of categories or globs (e.g., ‘UValue’, ‘Urbi.*’) *not* to display. See also **GD_ENABLE_CATEGORY**.

GD_LEVEL Set the verbosity level of traces. This environment variable is meant for the developers of Urbi SDK, yet it is very useful when tracking problems such as a UObject that fails to load properly. Valid values are, in increasing verbosity order:

1. **NONE**, no log messages at all.
2. **LOG**, the default value.
3. **TRACE**
4. **DEBUG**
5. **DUMP**, maximum verbosity.

GD_LOC If set, display the location (file, line, function name) from which the log message was issued.

GD_NO_COLOR If set, forbid the use of colors in the logs, even if the output device seems to support them, or if **GD_COLOR** is set.

GD_PID If set, display the PID (Process Identity).

GD_THREAD If set, display the thread identity.

GD_TIME If set, display timestamps.

GD_TIMESTAMP_US If set, display timestamps in microseconds.

The following variables control more high-level features, typically to override the default behavior.

URBI_PATH The search-path for urbiscript source files (i.e., ‘*.u’ files).

URBI_ROOT Urbi SDK is relocatable: its components know the relative location of each other. Yet they need to “guess” the *urbi-root*, i.e., the path to the directory that contains all the files. This variable permits to override that guess. Do not use it unless you know exactly what you are doing.

URBI_TEXT_MODE If set in the environment of a remote urbi-launch, disable binary protocol and force using urbiscript messages.

`URBI_UOBJECT_PATH` The search-path for UObjects files. This is used by `urbi-launch`, by `System.loadModule` and `System.loadLibrary`.

`URBI_UVAR_HOOK_CHANGED` If set, set `UVar.hookChanged` to false.

There are also very special environment variables, meant to be used only by Urbi developers, see [Section 21.8](#).

22.2 Special Files

‘CLIENT.INI’ This is the obsolete name for ‘`global.u`’.

‘`global.u`’ If found in the `URBI_PATH` (see [Section 22.1](#)), this file is loaded by Urbi server upon start-up. It is the appropriate place to install features you mean to provide to all the users of the server. It will be loaded via a special system connection, with its own private lobby. Therefore, purely local definitions will not be reachable from users; global modifications should be made in globally visible objects, say `Global`.

‘`local.u`’ If found in the `URBI_PATH` (see [Section 22.1](#)), this file is loaded by every connection established with an Urbi server. This is the appropriate place for enhancements local to a lobby.

‘URBI.INI’ This is the obsolete name for ‘`global.u`’.

22.3 urbi — Running an Urbi Server

The `urbi` program launches an Urbi server, for either batch, interactive, or network-based executions. It is subsumed by, but simpler to use than, `urbi-launch` ([Section 22.5](#)).

22.3.1 Options

General Options

‘`-h`, ‘`--help`’

Output the help message and exit successfully.

‘`--version`’

Output version information and exit successfully.

‘`--print-root`’

Output the `urbi-root` and exit successfully.

Tuning

‘`-d`, ‘`--debug=level`’

Set the verbosity level of traces. See the `GD_LEVEL` documentation ([Section 22.1.2](#)).

‘`-F`, ‘`--fast`’

Ignore system time, go as fast as possible. Do not use this option unless you know exactly what you are doing.

The ‘`--fast`’ flag makes the kernel run the program in “simulated time”, as fast as possible. A `sleep` in fast mode will not actually wait (from the wall-clock point of view), but the kernel will internally increase its simulated time.

For instance, the following session behaves equally in fast and non-fast mode:

```
{ sleep(2s); echo("after") } & { sleep(1s); echo("before") };
[000000463] *** before
[000001463] *** after
```

However, in non fast mode the execution will take two seconds (wall clock time), while it be instantaneous in fast mode. This option was designed for testing purpose; *it does not preserve the program semantics*.

`'-s', '--stack-size=size'`

Set the coroutine *stack size*. The unit of *size* is KB; it defaults to 128.

This option should not be needed unless you have “stack exhausted” messages from urbi in which case you should try ‘`--stack-size=512`’ or more.

Alternatively you can define the environment variable URBI_STACK_SIZE. The option ‘`--stack-size`’ has precedence over the URBI_STACK_SIZE.

`'-q', '--quiet'`

Do not send the welcome banner to incoming clients.

Networking

`'-H', '--host=address'`

Set the *address* on which network connections are listened to. Typical values of *address* include:

localhost only local connections are allowed (no other computer can reach this server).

127.0.0.1 same as localhost.

0.0.0.0 any IP v4 connection is allowed, including from remote computers.

Defaults to 0.0.0.0.

`'-P', '--port=port'`

Set the port to listen incoming connections to. If *port* is -1, no networking. If *port* is 0, then the system will chose any available port (see ‘`--port-file`’). Defaults to -1.

`'-w', '--port-file=file'`

When the system is up and running, and when it is ready for network connections, create the file named *file* which contains the number of the port the server listens to.

Execution

`'-e', '--expression=exp'`

Send the urbiscript expression *exp*. No separator is added, you have to pass yours.

`'-f', '--file=file'`

Send the contents of the file *file*. No separator is added, you have to pass yours.

`'-i', '--interactive'`

Start an interactive session.

Enabled if no input was provided (i.e., none of ‘`-e`’/‘`--expression`’, ‘`-f`’/‘`--file`’, ‘`-P`’/‘`-port`’ was given).

`'-m', '--module=module'`

Load the UObject *module*.

The options ‘`-e`’, ‘`-f`’ accumulate, and are run in the same [Lobby](#) as ‘`-i`’ if used. In other words, the following session is valid:

```
# Create a file "two.u".
$ echo "var two = 2;" >two.u
$ urbi -q -e 'var one = 1;' -f two.u -i
[00000000] 1
[00000000] 2
one + two;
[00000000] 3
```

22.3.2 Quitting

To exit `urbi`, use the command `System.shutdown`.

If you are in interactive mode, you can also use the shortcut sequence C-d in the server lobby. The command `Lobby.quit` does not shutdown the server, it only closes the current lobby which closes the connection while in remote mode (using `telnet` for example), but only closes the interactive mode if performed on the server lobby.

On Unix systems, `urbi` handles the SIGINT signal (action performed when C-c is pressed). If you are in interactive mode, a first C-c kills the foreground job (or does nothing if no job is running on foreground).

For example consider the following piece of code:

```
every (4s)
  echo("Hello world!");
```

urbiscript
Session

If you try to execute this code, you will notice that you can no longer execute commands, since the `every` job is foreground (because of the semicolon). Now if you press C-c, the `every` job is killed, all the pending commands you may have typed are cleared from the execution queue, and you get back with a working interactive mode.

```
every (1s) echo("Hello world!");
[00010181] *** Hello world!
[00011181] *** Hello world!
// This command is entered while the foreground job blocks the evaluation.
// It waits to be executed.
echo("done");
[00012181] *** Hello world!
[00013181] *** Hello world!

// Pressing Control-C here:
[00014100:error] !!! received interruption, killing foreground job.
// Note that the "echo" is not run, the command queue is flushed.

// New commands are honored.
echo("interrupted");
[00019709] *** interrupted
```

urbiscript
Session

A second C-c (or SIGINT) within the 1.5s after the first one tries to execute `System.shutdown`. This can take a while, in particular if you have remote UObjects, since a clean shutdown will first take care of disconnecting them cleanly.

```
// Two Control-C in a row.
[00493865:error] !!! received interruption, killing foreground job.
[00494672:error] !!! received interruption, shutting down.
```

urbiscript
Session

Pressing C-c a third time triggers the default behavior: killing the program in emergency, by-passing all the cleanups.

In non-interactive mode (after a `Lobby.quit` on the server lobby for example), the first C-c executes `System.shutdown`, and the second one triggers the default behavior.

22.4 urbi-image — Querying Images from a Server

Shell
Session

```
urbi-image option...
```

Connect to an Urbi server, and fetch images from it, for instance from its camera.

22.4.1 Options

General Options

'-h', '--help'

Output the help message and exit successfully.

'--version'

Output version information and exit successfully.

Networking

'-H', '--host=host'

Address to connect to.

'-P', '--port=port'

Port to connect to.

'--port-file=file'

Connect to the port contained in the file *file*.

Tuning

'-p', '--period=period'

Specify the period, in millisecond, at which images are queried.

'-F', '--format=format'

Select format of the image ('rgb', 'ycrcb', 'jpeg', 'ppm').

'-r', '--reconstruct'

Use reconstruct mode (for aibo).

'-j', '--jpeg=factor'

JPEG compression factor (from 0 to 100, defaults to 70).

'-d', '--device=device'

Query image on *device.val* (default: `camera`).

'-o', '--output=file'

Query and save one image to *file*.

'-R', '--resolution=resolution'

Select resolution of the image (0=biggest).

'-s', '--scale=factor'

Rescale image with given *factor* (display only).

22.5 urbi-launch — Running a UObject

The `urbi-launch` program launches an Urbi system. It is more general than `urbi` (Section 22.3): everything `urbi` can do, `urbi-launch` can do it too.

22.5.1 Invoking urbi-launch

`urbi-launch` launches UObjects, either in plugged-in mode, or in remote mode. Since UObjects can also accept options, the command line features two parts, separated by ‘`--`’:

```
urbi-launch [option...] module... [-- module-option...]
```

Shell
Session

The *modules* are looked for in the `URBI_UOBJECT_PATH`. The *module* extension (`'.so'`, or `'.dll'`) does not need to be specified.

Options

`'-h', '--help'`

Output the help message and exit successfully.

`'--version'`

Output version information and exit successfully.

`'--print-root'`

Output the `urbi-root` and exit successfully.

`'-c', '--customize=file'`

Start the Urbi server in *file*. This option is mostly for developers.

`'-d', '--debug=level'`

Set the verbosity level of traces. See the `GD_LEVEL` documentation (Section 22.1.2).

Mode selection

`'-p', '--plugin'`

Attach the *module* onto a currently running Urbi server (identified by *host* and *port*). This is equivalent to running `loadModule("module")` on the corresponding server.

`'-r', '--remote'`

Run the *modules* as separated processes, connected to a running Urbi server (identified by *host* and *port*) via network connection.

`'-s', '--start'`

Start an Urbi server with plugged-in *modules*. In this case, the *module-option* are exactly the options supported by `urbi`.

Networking `urbi-launch` supports the same networking options (`--host`, `--port`, `--port-file`) as `urbi`, see Section 22.3.

22.5.2 Examples

To launch a fresh server in an interactive session with the `UMachine` UObject compiled as the file ‘`test/machine.so`’ (or ‘`test/machine.dll`’ plugged in), run:

```
urbi-launch --start test/machine -- --interactive
```

Shell
Session

To start an Urbi server accepting connections on the local port 54000 from any remote host, with `UMachine` plugged in, run:

```
urbi-launch --start --host 0.0.0.0 --port 54000 test/machine
```

Shell
Session

Since `urbi-launch` in server mode is basically the same as running the `urbi` program, both programs are quit the same way (see Section 22.3.2).

22.6 urbi-launch-java — Running a Java UObject

Java UObject can only be run remotely, they cannot be “plugged”. So, while `urbi-launch-java` is really alike `urbi-launch`, it is not the same either. See also [Section 6.1.2](#) for a more tutorial-like introduction to `urbi-launch-java`.

22.6.1 Invoking `urbi-launch-java`

`urbi-launch-java` launches Java UObject, either in plugged-in mode, or in remote mode. Since UObject can also accept options, the command line features two parts, separated by ‘`--`’:

Shell Session

```
urbi-launch-java [option...] module... [-- module-option...]
```

The `modules` are looked for in the `URBI_UOBJECT_PATH`. The `module` extension (‘`.so`’, or ‘`.dll`’) does not need to be specified.

Options

`-h`, `--help`

Output the help message and exit successfully.

`--version`

Output version information and exit successfully.

`--print-root`

Output the `urbi-root` and exit successfully.

`-C`, `--check`

Exit successfully if and only if Java UObject is usable (it might not be compiled or installed).

`-d`, `--debug=level`

Set the verbosity level of traces. See the `GD_LEVEL` documentation ([Section 22.1.2](#)).

Java

`-c`, `--classpath=path`

Pass a colon-separated list of directories, JAR archives, and ZIP archives to search for class files.

Networking `urbi-launch-java` supports the same networking options (‘`--host`’, ‘`--port`’, ‘`--port-file`’) as `urbi`, see [Section 22.3](#).

22.7 urbi-ping — Checking the Delays with a Server

Shell Session

```
urbi-ping option... [host] [interval] [count]
```

Send “ping”s to an Urbi server, and report how long it took for “pong”s to come back.

22.7.1 Options

General Options

`-h`, `--help`

Output the help message and exit successfully.

`--version`

Output version information and exit successfully.

Networking

- `-H`, `--host=host`**
Address to connect to.
- `-P`, `--port=port`**
Port to connect to.
- `--port-file=file`**
Connect to the port contained in the file *file*.

Tuning

- `-c`, `--count=count`**
Number of pings to send. Pass 0 to send an unlimited number of pings. Defaults to 0.
- `-i`, `--interval=interval`**
Set the delays between pings, in milliseconds. Pass 0 to flood the connection. Defaults to 1000.

22.8 urbi-send — Sending urbiscipt Commands to a Server

```
urbi-send option...
```

Shell
Session

Connect to an Urbi server, and send commands or file contents to it. Stay connected, until server disconnection, or user interruption (such as C-c under a Unix terminal).

- `-e`, `--expression=script`**
Send *script* to the server.
- `-f`, `--file=file`**
Send the contents of *file* to the server.
- `-m`, `--module=module`**
Load the UObject *module*. This is equivalent to `'[style=varInString]—e 'loadModule("module");'`.
- `-h`, `--help`**
Output the help message and exit successfully.
- `-H`, `--host=host`**
Address to connect to.
- `-P`, `--port=port`**
Port to connect to.
- `--port-file=file`**
Connect to the port contained in the file *file*.
- `-Q`, `--quit`**
Disconnect from the server immediately after having sent all the commands. This is equivalent to `'-e 'quit;''` at the end of the options. This is inappropriate if code running in background is expected to deliver its result asynchronously: the connection will be closed before the result was sent.
Without this option, `urbi-send` prompts the user to hit C-c to end the connection.
- `--version`**
Output version information and exit successfully.

22.9 urbi-sound — Querying Sounds from a Server

Shell
Session

```
urbi-sound option...
```

Connect to an Urbi server, and record sounds from it, and play them on ‘/dev/dsp’. This is only supported on GNU/Linux. If ‘[’o]output is specified, the recording is saved instead of being played.

22.9.1 Options

General Options

‘-h’, ‘--help’

Output the help message and exit successfully.

‘--version’

Output version information and exit successfully.

Networking

‘-H’, ‘--host=host’

Address to connect to.

‘-P’, ‘--port=port’

Port to connect to.

‘--port-file=file’

Connect to the port contained in the file *file*.

Tuning

‘-d’, ‘--device=device’

Query image on *device.val* (default: `camera`).

‘-D’, ‘--duration=duration’

Specify, in seconds, how long the recording is performed.

‘-n’, ‘--no-header’

When saving the sound, do not include the WAV headers in the file.

‘-o’, ‘--output=file’

Save the sound into *file*.

22.10 umake — Compiling UObject Components

The `umake` programs builds loadable modules, UObject, to be later run using `urbi-launch` ([Section 22.5](#)). Using it is not mandatory: users familiar with their compilation tools will probably prefer using them directly. Yet `umake` makes things more uniform and simpler, at the cost of less control.

22.10.1 Invoking umake

Usage:

Shell
Session

```
umake option... file...
```

Compile the *file*. The *files* can be of different kinds:

- objects files ('*.o', '*.obj' and so forth) and linked into the result.
- libraries ('*.a') and linked into the result.
- source files ('*.cc', '*.cpp', '*.c', '*.C') are compiled.
- header files ('*.h', '*.hh', '*.hxx', '*.hpp') are *not* compiled, but used as dependencies: if a header file is changed, the next `umake` run will actually recompile.
- directories are recursively traversed, and files of the above types are gathered as if they were given on the command line.

There are several environment variables that `umake` reads:

`EXTRA_CPPFLAGS` Options passed to the preprocessor.

`EXTRA_CXXFLAGS` Options passed to the C++ compiler.

`EXTRA_LDFLAGS` Options passed to the linker.

The arguments of `umake` may also include assignments, i.e., arguments of the type '*var=val*'. The behavior depends on *var*, the name of the variable:

`EXTRA_CPPFLAGS` Appended to the options passed to the preprocessor.

`EXTRA_CXXFLAGS` Appended to the options passed to the C++ compiler.

`EXTRA_LDFLAGS` Appended to the options passed to the linker.

`VPATH` Appended to the `VPATH`, i.e., the list of directories in which the sources are looked for.

Otherwise, the argument is passed as is to `make`.

General options

`--debug`

Turn on shell debugging (`set -x`) to track `umake` problems.

`-h`, `--help`

Output the help message and exit successfully.

`-q`, `--quiet`

Produce no output except errors.

`-v`, `--version`

Output version information and exit successfully.

`-v`, `--verbose`

Report on what is done.

Compilation options

`--deep-clean`

Remove all building directories and exit.

`-c`, `--clean`

Clean building directory before compilation.

`-j`, `--jobs=jobs`

Specify the numbers of compilation commands to run simultaneously.

‘-l’, ‘--library’
 Produce a library, don’t link to a particular core.

‘-s’, ‘--shared’
 Produce a shared library loadable by any core.

‘-o’, ‘--output=*file*’
 Set the output file name.

‘-C’, ‘--core=*core*’
 Set the build type.

‘-H’, ‘--host=*host*’
 Set the destination host.

‘-m’, ‘--disable-automain’
 Do not add the `main` function.

--package=*pkg*
 Use `pkg-config` to retrieve compilation flags (`cflags` and `libs`). Fails if `pkg-config` does not know about the package *pkg*.

Compiler and linker options

‘-D*symbol*
 Forwarded to the preprocessor (i.e., added to the `EXTRA_CPPFLAGS`).

‘-I*path*
 Forwarded to the preprocessor (i.e., added to the `EXTRA_CPPFLAGS`).

‘-L*path*
 Forwarded to the linker (i.e., added to the `EXTRA_LDFLAGS`).

‘-l*lib*
 Forwarded to the linker (i.e., added to the `EXTRA_LDFLAGS`).

Developer options

‘-p’, ‘--prefix=*dir*’
 Set library files location.

‘-P’, ‘--param-mk=*file*’
 Set ‘`param.mk`’ location.

‘-k’, ‘--kernel=*dir*’
 Set the kernel location.

22.10.2 `umake` Wrappers

As a convenience for common `umake` usages, some wrappers are provided:

`umake-deepclean` — Cleaning

Clean the temporary files made by running `umake` with the same arguments. Same as ‘`umake --deep-clean`’.

`umake-shared` — Compiling Shared UObjects

Build a shared object to be later run using `urbi-launch` (Section 22.5). Same as ‘`umake --shared-library`’.

Chapter 23

urbiscript Language Reference Manual

23.1 Syntax

23.1.1 Characters, encoding

Currently urbiscript makes no assumptions about the encoding used in the programs, but the streams are handled as 8-bit characters.

While you are allowed to use whatever character you want in the string literals (especially using the binary escapes, [Section 23.1.6.6](#)), only plain ASCII characters are allowed in the program body. Invalid characters are reported, possibly escaped if they are not “printable”. If you enter UTF-8 characters, since they possibly span over several 8-bit characters, a single (UTF-8) character may be reported as several invalid (8-bit) characters.

```
#Été;  
[00048238:error] !!! syntax error: invalid character: '#'  
[00048239:error] !!! syntax error: invalid character: '\xc3'  
[00048239:error] !!! syntax error: invalid character: '\x89'  
[00048239:error] !!! syntax error: invalid character: '\xc3'  
[00048239:error] !!! syntax error: invalid character: '\xa9'
```

urbiscript
Session

23.1.2 Comments

Comments are used to document the code, they are ignored by the urbiscript interpreter. Both C++ comment types are supported.

- A `//` introduces a comment that lasts until the end of the line.
- A `/*` introduces a comment that lasts until `*/` is encountered. Comments nest, contrary to C/C++: if two `/*` are encountered, the comment will end after two `*/`, not one.

```
1; // This is a one line comment.  
[00000001] 1  
  
2; /* an inner comment */ 3;  
[00000002] 2  
[00000003] 3  
  
4; /* nested /* comments */ 5; */ 6;  
[00000004] 4  
[00000005] 6  
  
7  
/*
```

urbiscript
Session

```

/*
 * Multi-line.
 */
;

[00000006] 7

```

23.1.3 Synclines

While the interaction with an urbiscript kernel is usually performed via a network connection, programmers are used to work with files which have names, line numbers and so forth. This is most important in error messages. Since even loading a file actually means sending its content as if it were typed in the network session, in order to provide the user with meaningful locations in error messages, urbiscript features *synclines*, a means to change the “current location”, similarly to `#line` in C-like languages. This is achieved using special `//#` comments.

The following special comments are recognized only as a whole line. If some component does not match exactly the expected syntax, or if there are trailing items, the whole line is treated as a comment.

- `//#line line "file"`
Specify that the *next* line is from the file named *file*, and which line number is *line*. The current location (i.e., current file and line) is lost.
- `//#push line "file"`
Save the current location, and then behave as if `//#line` was used.
- `//#pop`
Restore the current location. `//#push` and `//#pop` must match.

23.1.4 Identifiers

Identifiers in urbiscript are composed of one or more alphanumeric or underscore (`_`) characters, not starting by a digit, i.e., identifiers match the `[a-zA-Z_][a-zA-Z0-9_]*` regular expression. Additionally, identifiers must not match any of the urbiscript reserved words¹ documented in [Section 23.1.5](#). Identifiers can also be written between simple quotes (`'`), in which case they may contain any character.

urbiscript
Session

```

var x;
var foobar51;
var this.a_name_with_underscores;
// Invalid.
// var 3x;
// obj.3x();

// Invalid because "if" is a keyword.
// var if;
// obj.if();
// However, keywords can be escaped with simple quotes.
var 'if';
var this.'else';

// Identifiers can be escaped with simple quotes
var '%x';
var '1 2 3';
var this.'[]';

```

¹ The only exception to this rule is `new`, which can be used as the method identifier in a method call.

23.1.5 Keywords

Keywords are reserved words that cannot be used as identifiers, for instance. They are listed in [Table 23.1](#).

23.1.6 Literals

23.1.6.1 Angles

Angles are floats (see [Section 23.1.6.4](#)) followed by an angle unit. They are simply equivalent to the same float, expressed in radians. For instance, `180deg` (180 degrees) is equal to `pi`. Available units and their equivalent are presented in [Table 23.2](#).

```
pi == 180deg;
pi == 200grad;
```

Assertion Block

23.1.6.2 Dictionaries

Literal *dictionaries* are represented with a comma-separated, potentially empty list of arbitrary associations enclosed in square brackets (`[]`), as shown in the listing below. Empty dictionaries are represented with an association arrow between the brackets to avoid confusion with empty lists. See [Dictionary](#) for more details.

Each association is composed of a key, which is represented by a string, an arrow (`=>`) and an expression.

```
[ => ]; // The empty dictionary
[00000000] [ => ]
["a" => 1, "b" => 2, "c" => 3];
[00000000] ["a" => 1, "b" => 2, "c" => 3]
```

urbiscript Session

23.1.6.3 Durations

Durations are floats (see [Section 23.1.6.4](#)) followed by a time unit. They are simply equivalent to the same float, expressed in seconds. For instance, `1s 1ms`, which stands for “one second and one millisecond”, is strictly equivalent to `1.0001`. Available units and their equivalent are presented in [Table 23.3](#).

```
1d == 24h;
0.5d == 12h;
1h == 60min;
1min == 60s;
1s == 1000ms;

1s == 1;
1s 2s 3s == 6;
1s 1ms == 1.001;
1ms 1s == 1.001;
```

Assertion Block

23.1.6.4 Floats

urbiscript supports the *scientific notation* for floating-point literals. See [Float](#) for more details. Examples include:

```
1 == 1;
1 == 1.0;
1.2 == 1.2000;
1.234e6 == 1234000;
1e+11 == 1E+11;
```

Assertion Block

Keyword	Remark	Keyword	Remark
and	Synonym for <code>&&</code>	long	Reserved
and_eq	Synonym for <code>&=</code>	loop	<code>loop&</code> and <code>loop </code> flavors
asm	Reserved	loopn	Deprecated, use <code>for</code>
at		mutable	Reserved
auto	Reserved	namespace	Reserved
bitand	Synonym for <code>&</code>	new	
bitor	Synonym for <code> </code>	not	
bool	Reserved	not_eq	Synonym for <code>!=</code>
break		onleave	Synonym for <code> </code>
call		or	Synonym for <code> =</code>
case		or_eq	
catch		private	Ignored
char	Reserved	protected	Ignored
class		public	Ignored
closure	Synonym for <code>~</code>	register	Reserved
compl		reinterpret_cast	Reserved
const	Reserved	return	
const_cast		short	Reserved
continue		signed	Reserved
default		sizeof	Reserved
delete	Reserved	static	Deprecated
do		static_cast	Reserved
double	Reserved	stopif	
dynamic_cast	Reserved	struct	
else		switch	
emit	Deprecated	template	
enum		this	
every		throw	
explicit	Reserved	timeout	
export	Reserved	try	
extern	Reserved	typedef	
external		typeid	
float	Reserved	typename	
for	<code>for&</code> and <code>for </code> flavors	union	
foreach	Deprecated, use <code>for</code>	unsigned	
freezeif		using	
friend		var	
function	Reserved	virtual	
goto		volatile	
if		waituntil	
in		wchar_t	
inline	Reserved	whenever	
int	Reserved	while	<code>while&</code> and <code>while </code> flavors
internal	Deprecated	xor	Synonym for <code>^</code>
		xor_eq	Synonym <code>^=</code>

Table 23.1: Keywords

unit	abbreviation	equivalence for n
radian	rad	n
degree	deg	$n/180 * \pi$
grad	grad	$n/200 * \pi$

Table 23.2: Angle units

unit	abbreviation	equivalence for n
millisecond	ms	$n/1000$
second	s	n
minute	min	$n \times 60$
hour	h	$n \times 60 \times 60$
day	d	$n \times 60 \times 60 \times 24$

Table 23.3: Duration units

```
1e10 == 10000000000;
1e30 == 1e10 * 1e10 * 1e10;
```

Numbers are displayed rounded by the top level, but internally, as seen above, they keep their accurate value.

```
0.000001;
[00000011] 1e-06

0.0000001;
[00000012] 1e-07

0.00000000001;
[00000013] 1e-11

1e+3;
[00000014] 1000

1E-5;
[00000014] 1e-05
```

urbiscript
Session

In order to make numbers with units ('1min') and calling a method on a number ('1.min'), numbers that include a period must have a fractional part. In other words, '1.', if not followed by digits, is always read as '1 .':

```
1.;

[00004701:error] !!! syntax error: unexpected ;
```

urbiscript
Session

Hexadecimal notation is supported for integers: 0x followed by one or more hexadecimal digits, whose case is irrelevant.

```
0x2a == 42;
0x2A == 42;
0xabcdef == 11259375;
0xABCD E == 11259375;
0xFFFFFFFF == 4294967295;
```

Assertion
Block

Numbers with unknown suffixes are invalid tokens:

```
123foo;
[00005658:error] !!! syntax error: invalid token: '123foo'
12.3foo;
[00018827:error] !!! syntax error: invalid token: '12.3foo'
0xabcdef;
[00060432] 11259375
0xabcdefg;
[00061848:error] !!! syntax error: invalid token: '0xabcdefg'
```

urbiscript
Session

23.1.6.5 Lists

Literal *lists* are represented with a comma-separated, potentially empty list of arbitrary expressions enclosed in square brackets ([]), as shown in the listing below. See [List](#) for more details.

urbiscript
Session

```
[]; // The empty list
[00000000] []
[1, 2, 3];
[00000000] [1, 2, 3]
```

23.1.6.6 Strings

String literals are enclosed in double quotes ("") and can contain arbitrary characters, which stand for themselves, with the exception of the escape character, backslash (\), see below. The escapes sequences are defined in [Table 23.4](#).

\\"	backslash
\"	double-quote
\a	bell ring
\b	backspace
\f	form feed
\n	line feed
\r	carriage return
\t	tabulation
\v	vertical tabulation
\[0-7]{1,3}	eight-bit character corresponding to a one-, two- or three-digit octal number. For instance, \0, \000 and 177. The matching is greedy: as many digits as possible are taken: \0, \000 are both resolved in the null character.
\x[0-9a-fA-F]{2}	eight-bit character corresponding to a two-digit hexadecimal number. For instance, 0xFF.
\B(<i>length</i>)(<i>content</i>)	binary blob. A <i>length</i> -long sequence of verbatim <i>content</i> . <i>length</i> is expressed in decimal. <i>content</i> is not interpreted in any way. The parentheses are part of the syntax, they are mandatory. For instance \B(2)(\B)

Table 23.4: String escapes

Assertion Block

```
// Special characters.
"\\"" == "\\\"";
"\\\" == "\\\\";

// ASCII characters.
"\a" == "\007"; "\a" == "\x07";
"\b" == "\010"; "\b" == "\x08";
"\f" == "\014"; "\f" == "\x0c";
"\n" == "\012"; "\n" == "\x0a";
"\r" == "\015"; "\r" == "\x0d";
"\t" == "\011"; "\t" == "\x09";
"\v" == "\013"; "\v" == "\x0b";

// Octal escapes.
"\0" == "\00"; "\0" == "\000";
"\0000" == "\0" "0";
"\062\063" == "23";

// Hexadecimal escapes.
"\x00" == "\0";
"\x32\x33" == "23";

// Binary blob escape.
"\B(3)("\\") == "\\" "\\\\";
```

Consecutive string literals are glued together into a single string. This is useful to split large strings into chunks that fit usual programming widths.

```
"foo" "bar" "baz" == "foobarbaz";
```

Assertion Block

The interpreter prints the strings escaped; for instance, line feed will be printed out as `\n` when a string result is dumped and so forth. An actual line feed will of course be output if a string content is printed with echo for instance.

```
";  
[00000000] ""  
"foo";  
[00000000] "foo"  
"a\nb"; // urbscript escapes string when dumping them  
[00000000] "a\nb"  
echo("a\nb"); // We can see there is an actual line feed  
[00000000] *** a  
b  
echo("a\nb");  
[00000000] *** a\nb
```

urbscript Session

See [String](#) for more details.

23.1.6.7 Tuples

Literal *tuples* are represented with a comma-separated, potentially empty list of arbitrary elements enclosed in parenthesis `()`, as shown in the listing below. One extra comma can be added after the last element. To avoid confusion between a 1 member Tuple and a parenthesized expression, the extra comma must be added. See [Tuple](#) for more details.

```
();  
[00000000] ()  
(1,);  
[00000000] (1,)  
(1, 2);  
[00000000] (1, 2)  
(1, 2, 3, 4,);  
[00000000] (1, 2, 3, 4)
```

urbscript Session

23.1.6.8 Pseudo classes

Objects meant to serve as prototypes are best defined using the [class](#) construct. See also the tutorial, [Section 12.4](#).

```
<class-statement>  
 ::= "class" <lvalue> (: <prototypes>)? <block>  
  
<lvalue>  
 ::= (<expression> ".")* "identifier"  
  
<prototypes>  
 ::= (<expression> ",")* expression  
  
<block>  
 ::= "{" <statement>* "}"
```

Grammar Excerpt

This results in the (constant) definition of the name *lvalue* in the current context ([class](#) construct can be used inside a scope or in an object) with:

- a slot named *type* which is a the trailing component of *lvalue* as a [String](#);
- a slot named *as type* that returns [this](#).

- the list of prototypes is equal to the list *prototypes* that served as parent, defaulting to `Object`;
- the *block* is evaluated in the context of this object, as with a `do`-block;
- the value of the whole statement is the newly defined object.

```
urbiscript
Session
class Base
{
    var slot = 12;
}|;

assert
{
    hasLocalSlot("Base");
    Base.type == "Base";
    Base.protos == [Object];
    Base.slot == 12;
    Base.asBase === Base;
};

class Global.Derive : Base
{
    var slot = 34;
}|;

assert
{
    Global.hasLocalSlot("Derive");
    Global.Derive.type == "Derive";
    Global.Derive.protos == [Base];
    Global.Derive.slot == 34;
    Global.Derive.asDerive === Global.Derive;
    Global.Derive.asBase === Global.Derive;
};

class Base2 {}|;

class Derive2 : Base, Base2 {}|;

assert
{
    Derive2.type == "Derive2";
    Derive2.protos == [Base, Base2];
    Derive2.slot == 12;
    Derive2.asDerive2 === Derive2;
    Derive2.asBase === Derive2;
    Derive2.asBase2 === Derive2;
};
```

It is guaranteed that the expressions that define the class name and its parents are evaluated only once.

```
urbiscript
Session
function Global.verboseId(var x)
{
    echo(x) | x
}|;
class verboseId(Global).math : verboseId(Math)
{
};
[00000686] *** Global
[00000686] *** Math
[00000686] math
```

23.1.7 Statement Separators

Sequential languages such as C++ support a single way to compose two statements: the sequential composition, “denoted” by ‘;’. To support concurrency and more fine tuned sequentiality, urbiscript features four different statement-separators (or connectors):

‘;’ sequentiality

‘|’ tight sequentiality

‘,’ background concurrency

‘&’ fair-start concurrency

23.1.7.1 ‘;’

The ‘;’-connector waits for the first statement to finish before starting the second statement. When used in the top-level interactive session, both results are displayed.

```
1; 2; 3;
[00000000] 1
[00000000] 2
[00000000] 3
```

urbiscript
Session

23.1.7.2 ‘,’

The ‘,’-connector sends the first statement in background for concurrent execution, and starts the second statement when possible. When used in interactive sessions, the value of backgrounded statements are *not* printed — the time of their arrival being unpredictable, such results would clutter the output randomly. Use [Channels](#) or [Events](#) to return results asynchronously.

```
{
  for (3)
  {
    sleep(1s);
    echo("ping");
  },
  sleep(0.5s);
  for (3)
  {
    sleep(1s);
    echo("pong");
  },
}
[00000316] *** ping
[00000316] *** pong
[00000316] *** ping
[00000316] *** pong
[00000316] *** ping
[00000316] *** pong
```

urbiscript
Session

Both ‘;’ and ‘,’ have equal precedence. They are scoped too: the execution follow “waits” for the end of the jobs back-grounded with ‘,’ before proceeding. Compare the two following executions.

```
{
  sleep(100ms) | echo("1"),
  sleep(400ms) | echo("2"),
  echo("done");
}
[00000316] *** done
[00000316] *** 1
[00000316] *** 2
```

urbiscript
Session

urbiscript
Session

```
{
    sleep(100ms) | echo("1"),
    sleep(400ms) | echo("2"),
};

echo("done");
[00000316] *** 1
[00000316] *** 2
[00000316] *** done
```

23.1.7.3 ‘|’

When using the ‘;’ connector, the scheduler is allowed to run other commands between the first and the second statement. The ‘|’ does not yield between both statements. It is therefore more efficient, and, in a way, provides some atomicity for concurrent tasks.

urbiscript
Session

```
{
    { echo("11") ; sleep(100ms) ; echo("12") },
    { echo("21") ; sleep(400ms) ; echo("22") },
};

[00000316] *** 11
[00000316] *** 21
[00000316] *** 12
[00000316] *** 22
```

urbiscript
Session

```
{
    { echo("11") | echo("12") },
    { echo("21") | echo("22") },
};

[00000316] *** 11
[00000316] *** 12
[00000316] *** 21
[00000316] *** 22
```

In an interactive session, both statements must be “known” before launching the sequence. The value of the composed statement is the value of the second statement.

23.1.7.4 ‘&’

The ‘&’ is very similar to the ‘,’ connector, but for its precedence. Urbi expects to process the whole statement before launching the connected statements. This is especially handy in interactive sessions, as a means to fire a set of tasks concurrently.

23.1.8 Operators

urbiscript supports many *operators*, most of which are inspired from C++. Their syntax is presented here, and they are sorted and described with their original semantics — that is, + is an arithmetic operator that sums two numeric values. However, as in C++, these operators might be used for any other purpose — that is, + can also be used as the concatenation operator on lists and strings. Their semantics is thus not limited to what is presented here.

Tables in this section sort operators top-down, by precedence order. Group of rows (not separated by horizontal lines) describe operators that have the same precedence. Many operators are syntactic sugar that bounce on a method. In this case, the equivalent desugared expression is shown in the “Equivalence” column. This can help you determine what method to override to define an operator for an object (see [Section 12.6](#)).

This section defines the syntax, precedence and associativity of the operators. Their semantics is described in [Chapter 24](#) in the documentation of the classes that provide them.

23.1.8.1 Arithmetic operators

urbiscript supports classic *arithmetic operators*, with the classic semantics on numeric values. See [Table 23.5](#) and the listing below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
+	+a	-	Identity	a.'+'()
-	-a	-	Opposite	a.'-'()
**	a ** b	Right	Exponentiation	a.'**'(b)
*	a * b	Left	Multiplication	a.'*'(b)
/	a / b	Left	Division	a.'/'(b)
%	a % b	Left	Modulo	a.'%'(b)
+	a + b	Left	Sum	a.'+'(b)
-	a - b	Left	Difference	a.'-'(b)

Table 23.5: Arithmetic operators

```
1 + 1 == 2;
1 - 2 == -1;
2 * 3 == 6;
10 / 2 == 5;
2 ** 10 == 1024;
-(1 + 2) == -3;
1 + 2 * 3 == 7;
(1 + 2) * 3 == 9;
-2 ** 2 == -4;
----- 1 == 1;
```

Assertion Block

23.1.8.2 Assignment operators

Assignment in urbiscript can be performed with the `=` operator. Assignment operators, such as `+=`, are supported too, see [Table 23.6](#) and the examples below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
=	a = b	Right	Assignment	updateSlot("a", b) ²
+=	a += b	Right	In place assignment	a = a.'+='(b)
-=	a -= b	Right	In place assignment	a = a.'-='(b)
*=	a *= b	Right	In place assignment	a = a.'*='(b)
/=	a /= b	Right	In place assignment	a = a.'/='(b)
%=	a %= b	Right	In place assignment	a = a.'%='(b)
^=	a ^= b	Right	In place assignment	a = a.'^='(b)

Table 23.6: Assignment operators

The following example demonstrates that `a += b` behaves as `a = a + b` for Floats.

```
var y = 0;
[00000000] 0
y = 10;
[00000000] 10
y += 10;
[00000000] 20
y /= 5;
[00000000] 4
```

urbiscript Session

²For object fields only. Assignment to local variables cannot be redefined.

These operators are redefinable. Indeed, `a += b` is actually processed as `a = a.'+='(b)`. This definition, which is neither that of C (`a = a.'+'(b)`) nor that of C++ (`a.'+='(b)`), provides support for both *immutable* and *mutable* values.

Immutable Values Small objects such as Floats should typically be immutable, i.e., the value of a Float cannot change:

```
urbiscript
Session
var value = 0|;
var valueAlias = value|;
value += 10;
[00002275] 10
valueAlias;
[00002301] 0
```

It would be traitorous for most users that `valueAlias` be equal to 10 too. That's why `Float.'+='` (which is actually `Object.'+='`) simply bounces to `Float.'+'`. The "net result" of `value += 10` is therefore `value = value.'+'(10)`, i.e., a *new* Float is computed from `0.'+'(10)`, and `value` is rebound to it. The binding from `valueAlias` to 0 is left as is.

Mutable Values On the contrary, large, "updatable" objects should provide an implementation of `'+='` that mutates them. For instance, implementing `a.'+='(b)` as `a.'+'(b)` would be too costly for [Lists](#). Each time `+=` is used, we need to create a new List (whose content is that of `a`), then to append the contents of `b`, and finally throw away the former value of `a`.

Not only is this inefficient, this is also wrong (at least from a certain point of view). Indeed, since we no longer update the List pointed to by `a`, but rather store a new List, everything that was special to the original List (its uid or whatever special slot the user may have defined) is lost. The proper implementation of `List.'+='` is therefore to *modify this* by appending the added members.

```
urbiscript
Session
var myList = []|;
var myList.specialFeature = 42|;
myList += [1, 2, 3];
[00848865] [1, 2, 3]
myList.specialFeature;
[00848869] 42
var myOtherList = myList + [4, 5];
[00848873] [1, 2, 3, 4, 5]
myOtherList.specialFeature;
[00848926:error] !!! lookup failed: specialFeature
```

Note however that this means that because `a += b` is *not* processed as `a = a + b`, aliases to `a` are possibly modified.

```
urbiscript
Session
var something = []|;
var somethingElse = something|;
something += [1, 2];
[00008557] [1, 2]
somethingElse += [3, 4];
[00008562] [1, 2, 3, 4]
something;
[00008566] [1, 2, 3, 4]
```

Example So basically, the rules to redefine these operators are:

Immutable (small) objects `'+='` should redirect to `'+'` (which of course should *not* modify its target).

Mutable (large) objects `'+='` should update `this` and return it.

The following examples contrasts both approaches.

urbiscript
Session

```
class Counter
{
    var count = 0;
    function init (n) { var this.count = n };
    // Display the value, and the identity.
    function asString() { "%s @ %s" % [count, uid] };
    function '+'(var n) { new(count + n) };
    function '-'(var n) { new(count - n) };
};
```

```
class ImmutableCounter : Counter
{
    function '+='(var n) { this + n };
    function '-='(var n) { this - n };
};

var ic1 = ImmutableCounter.new(0);
[00010566] 0 @ 0x100354b70
var ic2 = ic1;
[00010574] 0 @ 0x100354b70

ic1 += 1;
[00010588] 1 @ 0x10875bee0

// ic1 points to a new object.
ic1;
[00010592] 1 @ 0x10875bee0
// ic2 still points to its original value.
ic2;
[00010594] 0 @ 0x100354b70
```

```
class MutableCounter : Counter
{
    function '+='(var n) { count += n | this };
    function '-='(var n) { count -= n | this };
};

var mc1 = MutableCounter.new(0);
[00029902] 0 @ 0x100364e00
var mc2 = mc1;
[00029911] 0 @ 0x100364e00

mc1 += 1;
[00029925] 1 @ 0x100364e00

// mc1 points to the same, updated, object.
mc1;
[00029930] 1 @ 0x100364e00
// mc2 too.
mc2;
[00029936] 1 @ 0x100364e00
```

23.1.8.3 Postfix Operators

In the tradition of C, urbiscript provides postfix operators (see [Table 23.7](#)), e.g., $b = a++$. Prefix operators, however, are not supported. Rather than $++a$, write $a += 1$.

Oper.	Syntax	Assoc.	Semantics	Equivalence
$++$	$a++$	-	Incrementation	$\{ \text{var } '$a' = a \mid a = a.'++' \mid '$a'\}$
$--$	$a--$	-	Decrementation	$\{ \text{var } '$a' = a \mid a = a.'--' \mid '$a'\}$

Table 23.7: Postfix operators

These operators *modify* the variable/slot they are applied to, and return the *former* value of the variable/slot.

```
{
    var count = 0;
    var countAlias = count;
    assert
    {
        count++ == 0;
        count == 1;
        countAlias == 0;
        count++ == 1;
        count == 2;
        countAlias == 0;
        count-- == 2;
        count == 1;
    };
};
```

urbiscript
Session

```
};
```

Similarly to assignment operators, these operators are redefinable. Indeed, `a++` is actually processed like `{ var '$save' = a | a = a.'++' | '$save' }`. In other words, you are entitled to redefine the operator `'++'` whose semantics is “return the successor of `this`”.

Beware that the operator `'++'` should *not* modify its target, but rather return a fresh value. Indeed, if it alters `this`, the copy made in `'$save'` will also have its value updated. In other words, the value of `a++` would be its new one, not its former one.

Redefining `'++'` is still an experimental feature which might be changed in future releases of Urbi SDK, do not rely on it.

23.1.8.4 Bitwise operators

urbiscript features *bitwise operators*. They are also used for other purpose than bit-related operations. See [Table 23.8](#) and the listing below.

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code><<</code>	<code>a << b</code>	Left	Left bit shift	<code>a.'<<'(b)</code>
<code>>></code>	<code>a >> b</code>	Left	Right bit shift	<code>a.'>>'(b)</code>
<code>^</code>	<code>a ^ b</code>	Left	Bitwise exclusive or	<code>a.'^'(b)</code>

Table 23.8: Bitwise operators

Assertion Block

```
4 << 2 == 16;
4 >> 2 == 1;
```

23.1.8.5 Logical operators

urbiscript supports the usual *Boolean operators*. See the table and the listing below. The operators `&&` and `||` are short-circuiting: their right-hand side is evaluated only if needed.

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code>!</code>	<code>!a</code>	Left	Logical negation	<code>a.'!'()</code>
<code>&&</code>	<code>a&&b</code>	Left	Logical and	<code>if (a) b else a</code>
<code> </code>	<code>a b</code>	Left	Logical or	<code>if (a) a else b</code>

Table 23.9: Boolean operators

The operator `!` returns the Boolean that is the negation of the value of its operand. See [Object.'!'](#).

Assertion Block

```
!true === false; !false === true;
!42 === false; !0 === true;
!"42" === false; !" === true;
![42] === false; ![] === true;
!"4"=>2] === false; ![=>] === true;
```

The operator `&&`, the short-circuiting logical and, behaves as follows. If the left-hand side operand evaluates to a “true” value, return the evaluation of the right-hand side operand; otherwise return the value of the left-hand side operand (not necessarily `false`).

Assertion Block

```
true && true;

(0 && "foo") == 0;
(2 && "foo") == "foo";

("") && "foo") == "";
("foo" && "bar") == "bar";
```

Its arguments are evaluated at most once.

```
var zero = 0|;
var one = 1|;
var two = 2|;

// First argument evaluated once, second is not needed.
({ echo("lhs") | zero } && { echo("rhs") | one }) === zero;
[00029936] *** lhs
[00029936] true

({ echo("lhs") | one } && { echo("rhs") | two }) === two;
[00029966] *** lhs
[00029966] *** rhs
[00029966] true
```

urbiscript
Session

The operator `||`, the short-circuiting logical or, behaves as follows. If the left-hand side operand evaluates to a “false” value, return the evaluation of the right-hand side operand; otherwise return the value of the left-hand side argument (not necessarily `true`).

```
true || false;

(0 || "foo") == "foo";
(2 || 1/0) == 2;

("") || "foo" == "foo";
("foo" || 1/0) == "foo";
```

Assertion
Block

Its arguments are evaluated at most once.

```
var zero = 0|;
var one = 1|;
var two = 2|;

// First argument evaluated once, second is not needed.
({ echo("lhs") | one } || { echo("rhs") | two }) === one;
[00029936] *** lhs
[00029936] true

({ echo("lhs") | zero } || { echo("rhs") | one }) === one;
[00029966] *** lhs
[00029966] *** rhs
[00029966] true
```

urbiscript
Session

See [Section 24.3.3](#) for more information about “true” and “false” values.

23.1.8.6 Comparison operators

urbiscript supports classical *comparison operators*. See [Table 23.10](#) and the listing below.

```
assert
{
  ! (0 < 0);
  0 <= 0;
  0 == 0;
  0 !== 0;
};

var z = 0;
[00000000] 0
assert
{
  z === z;
  ! (z !== z);
};
```

urbiscript
Session

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code>==</code>	<code>a == b</code>	None	Equality	<code>a.'=='(b)</code>
<code>!=</code>	<code>a != b</code>	None	Inequality	<code>a.'!='(b)</code>
<code>==></code>	<code>a ==> b</code>	None	Physical equality	<code>a.'==>'(b)</code>
<code>!==></code>	<code>a !==> b</code>	None	Physical inequality	<code>a.'!==>'(b)</code>
<code>~=</code>	<code>a ~= b</code>	None	Relative approximate equality	<code>a.'~='(b)</code>
<code>=~=</code>	<code>a =~= b</code>	None	Absolute approximate equality	<code>a.'=~='(b)</code>
<code><</code>	<code>a < b</code>	None	Less than	<code>a.'<'(b)</code>
<code><=</code>	<code>a <= b</code>	None	Less than or equal to	<code>a.'<='(b)</code>
<code>></code>	<code>a > b</code>	None	Greater than	<code>a.'>'(b)</code>
<code>>=</code>	<code>a >= b</code>	None	Greater than or equal to	<code>a.'>='(b)</code>

Table 23.10: Comparison operators

23.1.8.7 Container operators

These operators work on containers and their members. See [Table 23.11](#).

Oper.	Syntax	Assoc.	Semantics	Equivalence
<code>in</code>	<code>a in b</code>	-	Membership	<code>b.has(a)</code>
<code>not in</code>	<code>a not in b</code>	-	Non-membership	<code>b.hasNot(a)</code>
<code>[]</code>	<code>a[args]</code>	-	Subscript	<code>a.'[]'(args)</code>
<code>[] =</code>	<code>a[args] = v</code>	-	Subscript assignment	<code>a.'[]='(args, v)</code>

Table 23.11: Container operators

The `in` and `not in` operators test the membership of an element in a container. They bounce to the container's `has` and `hasNot` methods (see [Container](#)). They are non-associative.

Assertion Block

```
1    in [0, 1, 2];
3 not in [0, 1, 2];

"one" in ["zero" => 0, "one" => 1, "two" => 2];
"three" not in ["zero" => 0, "one" => 1, "two" => 2];
```

The following operators use an index. Note that the *subscript* (square bracket) operator is *variadic*: it takes any number of arguments that will be passed to the `'[]'` method of the targeted object.

urbiscript Session

```
// On lists.
var l = [1, 2, 3, 4, 5];
[00000000] [1, 2, 3, 4, 5]
assert
{
  l[0] == 1;
  l[-1] == 5;
  (l[0] = 10) == 10;
  l == [10, 2, 3, 4, 5];
};

// On strings.
var s = "abcdef";
[00000005] "abcdef"
assert
{
  s[0] == "a";
  s[1,3] == "bc";
  (s[1,3] = "foo") == "foo";
  s == "afoodef";
};
```

23.1.8.8 Object operators

These core operators provide access to slots and their properties. See [Table 23.12](#).

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	-	Message sending	Not redefinable
.	a.b(args)	-	Message sending	Not redefinable
->	a->b	-	Property access	getProperty("a", "b")
->	a->b = v	-	Property assignment	setProperty("a", "b", v)
.&	a.&b	-	Slot access	a.getSlot("b")

Table 23.12: Object operators

```
var obj = Object.new();
function obj.f() { 24 };

assert
{
    obj.f == 24;
    obj.&f != 24;
    obj.&f.isA(&Code);
    obj.&f === obj.getSlot("f");
};
```

urbiscript
Session

23.1.8.9 All operators summary

[Table 23.13](#) is a summary of all operators, to highlight the overall precedences. Operators are sorted by decreasing precedence. Groups of rows represent operators with the same precedence.

23.2 Scopes and local variables

23.2.1 Scopes

Scopes are sequences of statements, enclosed in curly brackets ({}). Statements are separated with the four statements separators (see [Section 23.1.7](#)). A trailing ‘;’ or ‘,’ is ignored. A trailing ‘&’ or ‘|’ behaves as if & {} or | {} was used. This particular case is heavily used by urbiscript programmers to discard the value of an expression:

```
// Return value is 1. Displayed.
1;
[00000000] 1
// Return value is that of {}, i.e., void. Nothing displayed.
1 | {};
// Same as "1 | {}", a valueless expression.
1|;
```

urbiscript
Session

Scopes are themselves expressions, and can thus be used in composite expressions, nested, and so forth.

```
// Scopes evaluate to their last expression
{
    1;
    2;
    3; // This last separator is optional.
};
[00000000] 3
// Scopes can be used as expressions
{1; 2; 3} + 1;
[00000000] 4
```

urbiscript
Session

Oper.	Syntax	Assoc.	Semantics	Equivalence
.	a.b	-	Message sending	Not redefinable
.	a.b(args)	-	Message sending	Not redefinable
->	a->b	-	Property access	getProperty("a", "b")
->	a->b = v	-	Property assignment	setProperty("a", "b", v)
.&	a.&b	-	Slot access	a.getSlot("b")
[]	a[args]	-	Subscript	a.'[]'(args)
[] =	a[args] = v	-	Subscript assignment	a.'[]='(args, v)
+	+a	-	Identity	a.'+'()
-	-a	-	Opposite	a.'-'()
**	a ** b	Right	Exponentiation	a.'**'(b)
*	a * b	Left	Multiplication	a.'*(b)
/	a / b	Left	Division	a.'/'(b)
%	a % b	Left	Modulo	a.'%'(b)
+	a + b	Left	Sum	a.'+'(b)
-	a - b	Left	Difference	a.'-'(b)
<<	a << b	Left	Left bit shift	a.'<<'(b)
>>	a >> b	Left	Right bit shift	a.'>>'(b)
==	a == b	None	Equality	a.'=='(b)
!=	a != b	None	Inequality	a.'!='(b)
==>	a === b	None	Physical equality	a.'==='(b)
!==>	a !== b	None	Physical inequality	a.'!=='(b)
=~>	a =~= b	None	Absolute approximate equality	a.'=~='(b)
~=>	a ~= b	None	Relative approximate equality	a.'~='(b)
<	a < b	None	Less than	a.'<'(b)
<=	a <= b	None	Less than or equal to	a.'<='(b)
>	a > b	None	Greater than	a.'>'(b)
>=	a >= b	None	Greater than or equal to	a.'>='(b)
^	a ^ b	Left	Bitwise exclusive or	a.'^'(b)
!	!a	Left	Logical negation	a.'!'()
in	a in b	-	Membership	b.has(a)
not in	a not in b	-	Non-membership	b.hasNot(a)
&&	a&&b	Left	Logical and	if (a) b else a
	a b	Left	Logical or	if (a) a else b
=	a = b	Right	Assignment	updateSlot("a", b)
+=	a += b	Right	In place assignment	a = a.'+='(b)
-=	a -= b	Right	In place assignment	a = a.'-='(b)
*=	a *= b	Right	In place assignment	a = a.'*='(b)
/=	a /= b	Right	In place assignment	a = a.'/='(b)
%=	a %= b	Right	In place assignment	a = a.'%='(b)
^=	a ^= b	Right	In place assignment	a = a.'^='(b)
++	a++	-	Incrementation	{var '\$a' = a a = a.'++' '\$a'
--	a--	-	Decrementation	{var '\$a' = a a = a.'--' '\$a'

Table 23.13: Operators summary

23.2.2 Local variables

Local variables are introduced with the `var` keyword, followed by an identifier (see [Section 23.1.4](#)) and an optional initialization value assignment. If the initial value is omitted, it defaults to `void`. Variable declarations evaluate to the initialization value. They can later be referred to by their name. Their value can be changed with the assignment operator; such an assignment expression returns the new value. The use of local variables is illustrated below.

```
// This declare variable x with value 42, and evaluates to 42.
var t = 42;
[00000000] 42
// x equals 42
t;
[00000000] 42
// We can assign it a new value
t = 51;
[00000000] 51
t;
[00000000] 51
// Initialization defaults to void
var u;
u.isVoid;
[00000000] true
```

urbiscript
Session

The lifespan of local variables is the same as their enclosing scope. They are thus only accessible from their scope and its sub-scopes³. Two variables with the same name cannot be defined in the same scope. A variable with the same name can be defined in an inner scope, in which case references refer to the innermost variable, as shown below.

```
{
  var x = "x";
  var y = "outer y";
  {
    var y = "inner y";
    var z = "z";
    // We can access variables of parent scopes.
    echo(x);
    // This refers to the inner y.
    echo(y);
    echo(z);
  };
  // This refers to the outer y.
  echo(y);
  // This would be invalid: z does not exist anymore.
  // echo(z);
  // This would be invalid: x is already declared in this scope.
  // var x;
}
[00000000] *** x
[00000000] *** inner y
[00000000] *** z
[00000000] *** outer y
```

urbiscript
Session

23.3 Functions

23.3.1 Function Definition

Functions in urbiscript are first class citizens: a function is a value, like floats and strings, and can be handled as such. This is different from most C-like languages. One can create a functional value thanks to the `function` keyword, followed by the list of formal arguments and

³Local variables can actually escape their scope with lexical closures, see [Section 23.3.6](#).

a compound statement representing the body of the function. Formal arguments are a possibly-empty comma-separated list of identifiers. Non-empty lists of formal arguments may optionally end with a trailing comma. The listing below illustrates this.

urbiscript
Session

```
function () { echo(0) };
[00000000] function () { echo(0) }

function (arg1, arg2) { echo(0) };
[00000000] function (var arg1, var arg2) { echo(0) }

function (
    arg1, // Ignored argument.
    arg2, // Also ignored.
)
{
    echo(0)
};
[00000000] function (var arg1, var arg2) { echo(0) }
```

Usually functions are bound to an identifier to be invoked later. The listing below shows a short-hand to define a named function.

urbiscript
Session

```
// Functions are often stored in variables to call them later.
var f1 = function () {
    echo("hello")
}|
f1();
[00000000] *** hello

// This form is strictly equivalent, yet simpler.
function f2()
{
    echo("hello")
}|
f2();
[00000000] *** hello
```

Therefore, like regular values, functions can either be plain local variables or slots of objects. In the following example, initially the object `Foo` features neither a `foo` nor a `bar` slot, but its `init` function declares a *local* `foo` function, and a *slot* `bar`. The whole difference is the initial `this` in the definition of `bar` which makes it a slot, not a variable.

urbiscript
Session

```
class Foo
{
    function init()
    {
        // This is a function local to init().
        function foo() { 42 };
        function this.bar() { 51 };
        foo() + bar();
    };
}|

Foo.foo;
[00001720:error] !!! lookup failed: foo
Foo.bar;
[00001750:error] !!! lookup failed: bar

[00001787] 93
Foo.init;
Foo.foo;
[00001787:error] !!! lookup failed: foo
Foo.bar;
[00001818] 51
```

23.3.2 Arguments

The list of formal arguments defines the number of argument the function requires. They are accessible by their name from within the body. If the list of formal arguments is omitted, the number of effective arguments is not checked, and arguments themselves are not evaluated. Arguments can then be manipulated with the call message, explained below.

```
var f = function(a, b) {
    echo(b + a);
}
f(1, 0);
[00000000] *** 1
// Calling a function with the wrong number of argument is an error.
f(0);
[00000000:error] !!! f: expected 2 arguments, given 1
f(0, 1, 2);
[00000000:error] !!! f: expected 2 arguments, given 3
```

urbiscript
Session

Non-empty lists of effective arguments may end with an optional comma.

```
f(
    "bar",
    "foo",
);
[00000000] *** foobar
```

urbiscript
Session

23.3.3 Return value

The *return value* of the function is the evaluation of its body — that is, since the body is a scope, the last evaluated expression in the scope. Values can be returned manually with the `return` keyword followed by the value, in which case the evaluation of the function is stopped. If `return` is used with no value, the function returns `void`.

```
function g1(a, b)
{
    echo(a);
    echo(b);
    a // Return value is a
}
g1(1, 2);
[00000000] *** 1
[00000000] *** 2
[00000000] 1

function g2(a, b)
{
    echo(a);
    return a; // Stop execution at this point and return a
    echo(b); // This is not executed
}
g2(1, 2);
[00000000] *** 1
[00000000] 1

function g3()
{
    return; // Stop execution at this point and return void
    echo(0); // This is not executed
}
g3(); // Returns void, so nothing is printed.
```

urbiscript
Session

23.3.4 Call messages

Functions can access meta-information about how they were called, through a `CallMessage` object. The *call message* associated with a function can be accessed with the `call` keyword. It contains several information such as not-yet evaluated arguments, the name of the function, the target ...

23.3.5 Strictness

urbiscript features two different function calls: *strict* function calls, effective arguments are evaluated before invoking the function, and *lazy* function calls, arguments are passed as-is to the function. As a matter of fact, the difference is rather that there are strict functions and lazy functions.

Functions defined with a (possibly empty) list of formal arguments in parentheses are strict: the effective arguments are first evaluated, and then their value is given to the called function.

urbiscript
Session

```
function first1(a, b) {
    echo(a); echo(b)
}
first1({echo("Arg1"); 1},
       {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** Arg2
[00000000] *** 1
[00000000] *** 2
```

A function declared with no formal argument list is lazy. Use its call message to manipulate its arguments *not* evaluated. The listing below gives an example. More information about this can be found in the `CallMessage` class documentation.

urbiscript
Session

```
function first2
{
    echo(call.evalArgAt(0));
    echo(call.evalArgAt(1));
}
first2({echo("Arg1"); 1},
       {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** 1
[00000000] *** Arg2
[00000000] *** 2
```

A lazy function may implement a strict interface by evaluating its arguments and storing them as local variables, see below. This is less efficient than defining a strict function.

urbiscript
Session

```
function first3
{
    var a = call.evalArgAt(0);
    var b = call.evalArgAt(1);
    echo(a); echo(b);
}
first3({echo("Arg1"); 1},
       {echo("Arg2"); 2});
[00000000] *** Arg1
[00000000] *** Arg2
[00000000] *** 1
[00000000] *** 2
```

23.3.6 Lexical closures

Lexical closures are an additional scoping rule, with which a function can refer to a local variable located outside the function — but still in the current context. urbiscript supports read/write

lexical closures, meaning that the variable is shared between the function and the outer environment, as shown below.

```
var n = 0|
function c1()
{
    // x refers to a variable outside the function
    n++;
    echo(n);
}
c1();
[00000000] *** 1
n;
[00000000] 1
n++;
[00000000] 1
c1();
[00000000] *** 3
```

urbiscript
Session

The following listing illustrate that local variables can even escape their declaration scope by lexical closure.

```
function wrapper()
{
    // Normally, x is local to 'wrapper', and is limited to this scope.
    var x = 0;
    at (x > 1)
        echo("ping");
    // Here we make it escape the scope by returning a closure on it.
    return function() { x++ };
} |

var w = wrapper()|
w();
[00000000] 0
w();
[00000000] 1
[00000000] *** ping
```

urbiscript
Session

23.3.7 Variadic functions

Variadic functions are functions that take a variable number of arguments. They are created by using the `[]` tag after a formal argument: the function will accept any number of arguments, and they will be assigned to the variadic formal argument as a list.

```
function variadic(var args[])
{
    echo(args)
} |

variadic();
[00000000] ***
variadic(1, 2, 3);
[00000000] *** [1, 2, 3]
```

urbiscript
Session

There can be other formal arguments, as long as the variadic argument is at the last position. If `n` is the number of non variadic arguments, the function will request as least `n` effective arguments, which will be assigned to the non variadic arguments in order like a classical function call. All remaining arguments will be passed in list as the variadic argument.

```
function invalid(var args[], var last)
{} |
[00000000:error] !!! syntax error: argument after list-argument
```

urbiscript
Session

```

function variadic(var a1, var a2, var a3, var args[])
{
    echo(a1);
    echo(a2);
    echo(a3);
    echo(args)
} |

// Not enough arguments.
variadic();
[00000000:error] !!! variadic: expected at least 3 arguments, given 0

// No variadic arguments.
variadic(1, 2, 3);
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
[00000000] *** []

// Two variadic arguments.
variadic(1, 2, 3, 4, 5);
[00000000] *** 1
[00000000] *** 2
[00000000] *** 3
[00000000] *** [4, 5]

```

23.4 Objects

Any urbiscript value is an object. Objects contain:

- A set of slots, which associate an object to a name.
- A list of prototypes, which are also objects.

23.4.1 Slots

Objects can contain any number of *slots*, every slot has a name and a value. Slots are often called “fields”, “attributes” or “members” in other object-oriented languages.

23.4.1.1 Manipulation

The `Object.createSlot` function adds a slot to an object with the void `(??)` value. The `Object.updateSlot` function changes the value of a slot; `Object.getSlot` reads it. The `Object.setSlot` method creates a slot with a given value. Finally, the `Object.localSlotNames` method returns the list of the object slot’s name. The listing below shows how to manipulate slots. More documentation about these methods can be found in [Section 24.40](#).

urbiscript
Session

```

var o = Object.new|
assert (o.localSlotNames == []);

o.createSlot("test");
assert
{
    o.localSlotNames == ["test"];
    o.&test.isVoid;
};

o.updateSlot("test", 42);
[00000000] 42
assert

```

```
{
  o.&test == 42;
};
```

23.4.1.2 Syntactic Sugar

There is some syntactic sugar for slot methods:

- `var o.name` is equivalent to `o.createSlot("name")`.
- `var o.name = value` is equivalent to `o.setSlot("name", value)`.
- `o.name = value` is equivalent to `o.updateSlot("name", value)`.
- `o.&name` is equivalent to `o.getSlot("name")` (`o` can be omitted, like for regular method invocations: `&name` is equivalent to `getSlot("name")`).

23.4.2 Properties

Slots can have properties, see [Section 12.7](#) for an introduction to properties.

23.4.2.1 Manipulation

There is a number of functions to manipulate properties:

- `Object.setProperty`, to define/set a property.
- `Object.getProperty`, to get a property.
- `Object.removeProperty`, to delete a property.
- `Object.hasProperty`, to test for the existence of a property.
- `Object.properties`, to get all the properties of a slot.

There is also syntactic sugar for some of them:

- `slot->name` is equivalent to `getProperty("slot", "name")`.
- `slot->name = value` is equivalent to `setProperty("slot", "name", value)`.

23.4.2.2 Standard Properties

Some properties are handled by the system itself.

`changed` The `changed` property allows to monitor when a slot is bound to new values: each time a new value is assigned to the monitored slot, an event `changed` is emitted:

```
var x = []|;
at (x->changed?)
echo("x->changed");

x = [1]|;
[00092656] *** x->changed
x = [1, 2]|;
[00092756] *** x->changed
```

urbiscript
Session

Even if the slot is assigned to the very same value, the `x->changed` event is emitted.

urbiscript
Session

```
x = x|;
[00092856] *** x->changed
```

This is different from checking updates to the value a slot is bound to:

```
urbiscript
Session
x << 3;
[00092866] [1, 2, 3]
```

One can monitor updates using the `changed slot` (not property).

constant The `constant` property defines whether a slot can be assigned a new value.

```
urbiscript
Session
var c = 0;
[00000000] 0
c = 1;
[00000000] 1

c->constant = true;
[00000000] true
c = 2;
[00000000:error] !!! cannot modify const slot

c->constant = false;
[00000000] false
c = 3;
[00000000] 3
```

A new slot can be declared constant when first defined:

```
urbiscript
Session
const var two = 2;
[00000036] 2
two = 3;
[00000037:error] !!! cannot modify const slot
two->constant;
[00000038] true
```

23.4.3 Prototypes

23.4.3.1 Manipulation

urbiscript is a prototype-based language, unlike most classical object oriented language, which are class-based. In prototype-based languages, objects have no type, only *prototypes*, from which they inherit behavior.

urbiscript objects can have several prototypes. The list of prototypes is given by the `Object.protos` method; they can be added or removed with `Object.addProto` and `Object.removeProto`. See [Section 24.40](#) for more documentation.

```
urbiscript
Session
var ob = Object.new|
assert (ob.protos == [Object]);

ob.addProto(Pair);
[00000000] (nil, nil)
assert (ob.protos == [(nil, nil), Object]);

ob.removeProto(Object);
[00000000] (nil, nil)
assert (ob.protos == [(nil, nil)]);
```

23.4.3.2 Inheritance

Objects inherit their prototypes' slots: `getSlot` will also look in an object prototypes' slots. `getSlot` performs a depth-first traversal of the prototypes hierarchy to find slots. That is, when looking for a slot in an object:

- `getSlot` checks first if the object itself has the requested slot. If so, it returns its value.
- Otherwise, it applies the same research on every prototype, in the order of the prototype list (since `addProto` inserts in the front of the prototype list, the last prototype added has priority). This search is recursive: `getSlot` will also look in the first prototype's prototype, etc. before looking in the second prototype. If the slot is found in a prototype, it is returned.
- Finally, if no prototype had the slot, an error is raised.

The following example shows how slots are inherited.

```
var a = Object.new|
var b = Object.new|
var c = Object.new|
a.setSlot("x", "slot in a")|
b.setSlot("x", "slot in b")|
// c has no "x" slot
c.getSlot("x");
[00000000:error] !!! lookup failed: x
// c can inherit the "x" slot from a.
c.addProto(a)|
c.getSlot("x");
[00000000] "slot in a"
// b is prepended to the prototype list, and has thus priority.
c.addProto(b)|
c.getSlot("x");
[00000000] "slot in b"
// A local slot in c has priority over prototypes.
c.setSlot("x", "slot in c")|
c.getSlot("x");
[00000000] "slot in c"
```

urbiscript
Session

23.4.3.3 Copy on write

The `updateSlot` method has a particular behavior with respect to prototypes. Although performing an `updateSlot` on a non-existent slot is an error, it is valid if the slot is inherited from a prototype. In this case, the slot is however not updated in the prototype, but rather created in the object itself, with the new value. This process is called *copy on write*; thanks to it, prototypes are not altered when the slot is overridden in a child object.

```
var p = Object.new|
var p.slot = 0|
var d = Object.new|
d.addProto(p)|
d.slot;
[00000000] 0
d.slot = 1;
[00000000] 1
// p's slot was not altered
p.slot;
[00000000] 0
// It was copied in d
d.slot;
[00000000] 1
```

urbiscript
Session

23.4.4 Sending messages

A *message* in urbiscript consists in a message name and arguments. One can send a message to an object with the dot (.) operator, followed by the message name (which can be any valid identifier) and the arguments, as shown below. When there are no arguments, the parentheses can be omitted. As you might see, sending messages is very similar to calling methods in classical languages.

urbiscript
Session

```
// Send the message msg to object obj, with arguments arg1 and arg2.
obj.msg(arg1, arg2);
// Send the message msg to object obj, with no arguments.
obj.msg();
// This is strictly equivalent.
obj.msg;
```

When a message *msg* is sent to object *obj*:

- The *msg* slot of *obj* is retrieved (i.e., *obj.getSlot("msg")*). If the slot is not found, a lookup error is raised.
- If the object is a *routine* (either a primitive, written in C++ for instance, or a function implemented in urbiscript), it is invoked with the message arguments, and the returned value is the result. As a consequence, the number of arguments in the message sending must match the one required by the routine.
- Otherwise (the object is not a routine), this object is the result of the message sending. There must be no argument.

Such message sending is illustrated below.

urbiscript
Session

```
var obj = Object.new|
var obj.a = 42|
var obj.b = function (x) { x + 1 }|
obj.a;
[00000000] 42
obj.a();
[00000000] 42
obj.a(50);
[00000000:error] !!! a: expected 0 argument, given 1
obj.b;
[00000000:error] !!! b: expected 1 argument, given 0
obj.b();
[00000000:error] !!! b: expected 1 argument, given 0
obj.b(50);
[00000000] 51
```

23.5 Enumeration types

Enumeration types enable to create types represented by a finite set of values, like the `enum` declaration in C.

urbiscript
Session

```
enum Suit
{
    hearts,
    diamonds,
    clubs,
    spades, // Last comma is optional
};
```

Since everything is an object in urbiscript, enums are too, with [Enumeration](#) as prototype.

urbiscript
Session

```
Suit;
[00000001] Suit
Suit.protos;
[00000002] [Enumeration]
```

The possible enum values are stored inside the enum object. They inherit the enum object, so you can easily test whether an object is a Suit or not.

```
Suit.hearts;
[00000001] hearts
Suit.diamonds;
[00000002] diamonds
Suit.clubs.isA(Suit);
[00000003] true
42.isA(Suit);
[00000003] false
```

urbiscript
Session

Enumeration values support comparison and pattern matching. You can iterate on the enum object to cycle through all possible values.

```
function find_ace(var suit)
{
    switch (suit)
    {
        case Suit.spades: "The only card I need is";
        default:          "I have";
    }
};

for (var suit in Suit)
    echo("%s the ace of %s." % [find_ace(suit), suit]);
[00000001] *** I have the ace of hearts.
[00000002] *** I have the ace of diamonds.
[00000003] *** I have the ace of clubs.
[00000004] *** The only card I need is the ace of spades.
```

urbiscript
Session

23.6 Structural Pattern Matching

Structural *pattern matching* is useful to deconstruct tuples, lists and dictionaries with a small and readable syntax.

These patterns can be used in the following clauses:

- The left hand side of an assignment.
- `case`
- `catch`
- `at`
- `waituntil`
- `whenever`

The following examples illustrate the possibilities of *structural pattern matching* inside `case` clauses:

```
switch ( ("foo", [1, 2]) )
{
    // The pattern does not match the values of the list.
    case ("foo", [2, 1]):
        echo("fail");
```

urbiscript
Session

```
// The pattern does not match the tuple.
case ["foo", [1, 2]]:
    echo("fail");

// The pattern matches and binds the variable "l"
// but the condition is not verified.
case ("foo", var l) if l.size == 0:
    echo("fail");

// The pattern matches.
case ("foo", [var a, var b]):
    echo("foo(%s, %s)" % [a, b]);
};

[00000000] *** foo(1, 2)
```

23.6.1 Basic Pattern Matching

Matching is used in many locations and allows to match literal values (e.g., [List](#), [Tuple](#), [Dictionary](#), [Float](#), [String](#)). In the following expressions each pattern (on the left hand side) matches the value (on the right hand side).

urbiscript
Session

```
(1, "foo") = (1, "foo");
[00000000] (1, "foo")
[1, "foo"] = [1, "foo"];
[00000000] [1, "foo"]
["b" => "foo", "a" => 1] = ["a" => 1, "b" => "foo"];
[00000000] ["a" => 1, "b" => "foo"]
```

A [Exception.MatchFailure](#) exception is thrown when a pattern does not match.

urbiscript
Session

```
try
{
    (1, 2) = (3, 4)
}
catch (var e if e.isA(Exception.MatchFailure))
{
    e.message
};
[00000000] "pattern did not match"
```

23.6.2 Variable

Patterns can contain variable declarations, to match any value and to bind it to a new variable.

urbiscript
Session

```
{
    (var a, var b) = (1, 2);
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
{
    [var a, var b] = [1, 2];
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
{
    ["b" => var b, "a" => var a] = ["a" => 1, "b" => 2, "c" => 3];
    echo("a = %d, b = %d" % [a, b]);
};
[00000000] *** a = 1, b = 2
```

23.6.3 Guard

Patterns used inside a `switch`, a `catch` or an event catching construct accept guards.

Guard are used by appending a `if` after a pattern or after a matched event.

The following example is inspired from the `TrajectoryGenerator` where a `Dictionary` is used to set the trajectory type.

```
switch (["speed" => 2, "time" => 6s])
{
  case ["speed" => var s] if s > 3:
    echo("Too fast");
  case ["speed" => var s, "time" => var t] if s * t > 10:
    echo("Too far");
};
[00000000] *** Too far
```

urbiscript
Session

The same guard are available for `catch` statement.

```
try
{
  throw ("message", 0)
}
catch (var e if e.isA(Exception))
{
  echo(e.message)
}
catch ((var msg, var value) if value.isA(Float))
{
  echo("%s: %d" % [msg, value])
};
[00000000] *** message: 0
```

urbiscript
Session

Events catchers can have guards on the pattern arguments. You can add these inside `at`, `whenever` and `waituntil` statements.

```
{
  var e = Event.new;
  at (e?(var msg, var value) if value % 2 == 0)
    echo("%s: %d" % [msg, value]);

  // Does not trigger the "at" because the guard is not verified.
  e!("message", 1);

  // Trigger the "at".
  e!("message", 2);
};
[00000000] *** message: 2
```

urbiscript
Session

23.7 Imperative flow control

23.7.1 break

When encountered within a `for` or a `while` loop, `break` makes the execution jump after the loop.

```
var i = 5|
for (; true; echo(i))
{
  if (i > 8)
    break;
  i++;
};
[00000000] *** 6
```

urbiscript
Session

```
[00000000] *** 7
[00000000] *** 8
[00000000] *** 9
```

23.7.2 continue

When encountered within a `for` or a `while` loop, `continue` short-circuits the rest of the loop-body, and runs the next iteration (if there remains one).

urbiscript
Session

```
for (var i = 0; i < 8; i++)
{
    if (i % 2 != 0)
        continue;
    echo(i);
};

[00000000] *** 0
[00000000] *** 2
[00000000] *** 4
[00000000] *** 6
```

23.7.3 do

The `do` construct changes the target (`this`) when evaluating an expression. It is a convenient means to avoid repeating the same target several times.

urbiscript
Session

```
do (target)
{
    body
};
```

It evaluates `body`, with `this` being `target`, as shown below. The whole construct evaluates to the value of `body`.

urbiscript
Session

```
do (1024)
{
    assert(this == 1024);
    assert(sqrt == 32);
    setSlot("y", 23);
}.y;
[00000000] 23
```

23.7.4 if

As in most programming languages, conditionals are expressed with `if`.

urbiscript
Session

```
if (condition) then-clause
if (condition) then-clause else else-clause
```

First `condition` is evaluated; if it evaluates to a value which is true (Section 24.3.3), evaluate `then-clause`, otherwise, if applicable, evaluate `else-clause`.

urbiscript
Session

```
if (true) assert(true) else assert(false);
if (false) assert(false) else assert(true);
if (true) assert(true);
```

Beware that *there must not be a terminator after the then-clause*:

urbiscript
Session

```
if (true)
    assert(true);
else
    assert(false);
[00000002:error] !!! syntax error: unexpected else
```

Contrary to C/C++, it has value: it also implements the `condition ? then-clause : else-clause` construct. Unfortunately, due to syntactic constraints inherited from C, it is a *statement*: it cannot be used directly as an expression. But as everywhere else in urbiscript, to use a statement where an expression is expected, use braces:

```
assert(1 + if (true) 3 else 4 == 4);
[00000003:error] !!! syntax error: unexpected if
assert(1 + { if (true) 3 else 4 } == 4);
```

urbiscript
Session

The `condition` can be any statement list. Variables which it declares are visible in both the `then-clause` and the `else-clause`, but do not escape the `if` construct.

```
assert({if (false) 10 else 20} == 20);
assert({if (true) 10 else 20} == 10);

assert({if (true) 10 } == 10);

assert({if (var x = 10) x + 2 else x - 2} == 12);
assert({if (var x = 0) x + 2 else x - 2} == -2);

if (var xx = 123) xx | xx;
[00000005:error] !!! lookup failed: xx
```

urbiscript
Session

23.7.5 for

`for` comes in several flavors.

23.7.5.1 C-like for

urbiscript support the classical C-like `for` construct.

```
for (initialization; condition; increment)
  body
```

urbiscript
Session

It has the exact same behavior as C's `for`:

1. The `initialization` is evaluated.
2. `condition` is evaluated. If the result is false, executions jump after `for`.
3. `body` is evaluated. If `continue` is encountered, execution jumps to point 4. If `break` is encountered, executions jumps after the `for`.
4. The `increment` is evaluated.
5. Execution jumps to point 2.
6. The loop evaluates to `void`.

23.7.5.2 Range-for

urbiscript supports iteration over a collection with another form of the `for` loop.

```
for (var name : collection)
  body;
```

urbiscript
Session

It evaluates `body` for each element in `collection`. The loop evaluates to `void`. Inside `body`, the current element is accessible via the `name` local variable. The listing below illustrates this.

urbiscript
Session

```
for (var x : [0, 1, 2, 3, 4])
    echo(x.sqr);
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 9
[00000000] *** 16
```

This form of `for` simply sends the `each` message to *collection* with one argument: the function that takes the current element and performs `action` over it. Thus, you can make any object acceptable in a `for` by defining an adequate `each` method.

urbiscript
Session

```
var Hobbits = Object.new|
function Hobbits.each (action)
{
    action("Frodo");
    action("Merry");
    action("Pippin");
    action("Sam");
}
for (var name in Hobbits)
    echo("%s is a hobbit." % [name]);
[00000000] *** Frodo is a hobbit.
[00000000] *** Merry is a hobbit.
[00000000] *** Pippin is a hobbit.
[00000000] *** Sam is a hobbit.
// This for statement is equivalent to:
Hobbits.each(function (name) { echo("%s is a hobbit." % [name]) });
[00000000] *** Frodo is a hobbit.
[00000000] *** Merry is a hobbit.
[00000000] *** Pippin is a hobbit.
[00000000] *** Sam is a hobbit.
```

23.7.5.3 for n-times

urbiscript provides some support for simple replication of computations: it allow to repeat a loop body *n*-times. With the exception that the loop index is not available within the body, `for (n)` is equivalent to `for (var i: n)`. It supports the same flavors: `for;`, `for|`, and `for&`. The loop evaluates to `void`.

Assertion
Block

```
{ var res = []; for (3) { res << 1; res << 2 } ; res }
    == [1, 2, 1, 2, 1, 2];

{ var res = []; for|(3) { res << 1; res << 2 } ; res }
    == [1, 2, 1, 2, 1, 2];

{ var res = []; for&(3) { res << 1; res << 2 } ; res }
    == [1, 1, 1, 2, 2, 2];
```

Note that since these `for` loops are merely anonymous foreach-style loops, the argument needs not being an integer, any iterable value can be used.

Assertion
Block

```
3 == { var r = 0; for ([1, 2, 3]) r += 1; r};
3 == { var r = 0; for ("123")      r += 1; r};
```

23.7.6 if

urbiscript supports the usual `if` constructs.

urbiscript
Session

```
if (condition)
    action;
```

```
if (condition)
    action
else
    otherwise;
```

If the *condition* evaluation is true, *action* is evaluated. Otherwise, in the latter version, *otherwise* is executed. Contrary to C/C++, there *must not* be a semicolon after the *action*; it would end the `if/else` construct prematurely.

23.7.7 loop

Endless loops can be created with `loop`, which is equivalent to `while (true)`. The loop evaluates to `void`. Both sequential flavors, `loop;` and `loop;;`, are supported. The default flavor is `loop;;`.

```
{
    var n = 10|;
    var res = []|;
    loop;
    {
        n--;
        res << n;
        if (n == 0)
            break
    };
    res
}
===[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
```

Assertion Block

```
{
    var n = 10|;
    var res = []|;
    loop|
    {
        n--;
        res << n;
        if (n == 0)
            break
    };
    res
}
===[9, 8, 7, 6, 5, 4, 3, 2, 1, 0];
```

Assertion Block

23.7.8 switch

The `switch` statement in urbiscript is similar to C's one.

```
switch (value)
{
    case value_one:
        action_one;
    case value_two:
        action_two;
    //case ...:
    // ...
    default:
        default_action;
};
```

urbiscript Session

It might contain an arbitrary number of cases, and optionally a default case. The *value* is evaluated first, and then the result is compared sequentially with the evaluation of all cases

values, with the `==` operator, until one comparison is true. If such a match is found, the corresponding action is executed, and execution jumps after the `switch`. Otherwise, the default case — if any — is executed, and execution jumps after the switch. The switch itself evaluates to case that was evaluated, or to void if no match was found and there's no default case. The listing below illustrates `switch` usage.

Unlike C, there are no `break` to end `case` clauses: execution will never span over several cases. Since the comparisons are performed with the generic `==` operator, `switch` can be performed on any comparable data type.

urbiscript
Session

```
function sw(v)
{
    switch (v)
    {
        case "":
            echo("Empty string");
        case "foo":
            "bar";
        default:
            v[0];
    }
};

sw("");
[00000000] *** Empty string
sw("foo");
[00000000] "bar"
sw("foobar");
[00000000] "f"
```

23.7.9 while

The `while` loop is similar to C's one.

urbiscript
Session

```
while (condition)
    body;
```

If `condition` evaluation, is true, `body` is evaluated and execution jumps before the `while`, otherwise execution jumps after the `while`.

urbiscript
Session

```
var j = 3;
while (0 < j)
{
    echo(j);
    j--;
};
[00000000] *** 3
[00000000] *** 2
[00000000] *** 1
```

The default flavor for `while` is `while;`.

23.7.9.1 while;

The semantics of

urbiscript
Session

```
while; (condition)
    body;
```

is the same as

urbiscript
Session

```
condition | body ; condition | body ; ...
```

as long as `cond` evaluates to true, or until `break` is invoked. If `continue` is evaluated, the rest of the body is skipped, and the next iteration is started.

urbiscript
Session

```
{
  var i = 4|
  while (true)
  {
    i -= 1;
    echo ("in: " + i);
    if (i == 1)
      break
    else if (i == 2)
      continue;
    echo ("out: " + i);
  };
};

[00000000] *** in: 3
[00000000] *** out: 3
[00000000] *** in: 2
[00000000] *** in: 1
```

23.7.9.2 while—

The semantics of

```
while| (condition)
  body;
```

urbiscript
Session

is the same as

```
condition | body | condition | body | ...
```

urbiscript
Session

The execution is can be controlled by `break` and `continue`.

```
{
  var i = 4|
  while| (true)
  {
    i -= 1;
    echo ("in: " + i);
    if (i == 1)
      break
    else if (i == 2)
      continue;
    echo ("out: " + i);
  };
};

[00000000] *** in: 3
[00000000] *** out: 3
[00000000] *** in: 2
[00000000] *** in: 1
```

urbiscript
Session

23.8 Exceptions

23.8.1 Throwing exceptions

Use the `throw` keyword to *throw exceptions*, as shown below. Thrown exceptions will break the execution upward until they are caught, or until they reach the top-level — as in C++. Contrary to C++, exceptions reaching the top-level are printed, and won't abort the kernel — other and new connections will continue to execute normally.

```
throw 42;
[00000000:error] !!! 42
function inner() { throw "exn" } |
function outer() { inner() }|
```

urbiscript
Session

```
// Exceptions propagate to parent call up to the top-level
outer();
[00000000:error] !!! exn
[00000000:error] !!!      called from: inner
[00000000:error] !!!      called from: outer
```

23.8.2 Catching exceptions

Exceptions are *caught* with the `try/catch` construct. Its syntax is as follows:

Grammar Excerpt

```
<try-statement>
  ::= "try" "{" <statement>* "}" <catch-clause>+ <else-clause>? <finally-clause>?
    | "try" "{" <statement>* "}" <finally-clause>

<catch-clause>
  ::= "catch" "(" <pattern> ")" "{" <statement>* "}"
    | "catch" "{" <statement>* "}"

<else-clause>
  ::= "else" "{" <statement>* "}"

<finally-clause>
  ::= "finally" "{" <statement>* "}"
```

It consists of a first block of statements (the *try-block*), from which we want to catch exceptions, and one or more catch clauses to stop the exception (*catch-blocks*).

Each `catch` clause defines a pattern against which the thrown exception is matched. If no pattern is specified, the catch clause matches systematically (equivalent to `catch (...)` in C++). It is a syntax error if this catch-all clause is followed by a catch-clause with a pattern:

urbiscript Session

```
try {} catch {} catch (var e) {};
[00000701:error] !!! syntax error: catch: exception already caught by a previous clause
```

The catch-all clause, if present, must be last:

urbiscript Session

```
try {} catch (var e) {} catch {};
```

Exceptions thrown from the `try` block are matched sequentially against all catch clauses. The first matching clause is executed, and control jumps after the whole try/catch block. If no catch clause matches, the exception isn't stopped and continues upward.

urbiscript Session

```
function test(e)
{
  try
  { throw e; }
  catch (0)
  { echo("zero") }
  catch ([var x, var y])
  { echo(x + y) }
} | {};
test(0);
[00002126] *** zero
test([22, 20]);
[00002131] *** 42
test(51);
[00002143:error] !!! 51
[00002143:error] !!!      called from: test
```

If an `else`-clause is specified, it is executed if the `try` block did not raise an exception.

urbiscript Session

```
try { echo("try") }
catch { echo("catch") }
else { echo("else")};
[00002855] *** try
```

```
[00002855] *** else
try { echo("try"); echo("throw"); throw 0 }
catch { echo("catch")};
else { echo("else")};
[00002855] *** try
[00002855] *** throw
[00002855] *** catch
```

The value of the whole construct is:

- if the `try` block raised an exception
 - if the exception is not caught (or an exception is thrown from the catch-clause), then there is no value, as the control flow is broken;
 - if the exception is caught; then it's the value of the corresponding catch clause.

```
try { throw 0; "try" } catch (0) { "catch" } else { "else" };
[00467080] "catch"
```

urbiscript
Session

- if the `try` block finish properly, then

- if there is an `else`-clause, its value.

```
try { "try" } catch (0) { "catch" } else { "else" };
[00467080] "else"
```

urbiscript
Session

- otherwise the value of the `try`-block.

```
try { "try" } catch (0) { "catch" };
[00467080] "try"
```

urbiscript
Session

23.8.3 Inspecting exceptions

An `Exception` is a regular object, on which introspection can be performed.

```
try
{
  Math.cos(3,1415);
}
catch (var e)
{
  echo ("Exception type: %s" % e.type);
  if (e.isA(Exception.Arity))
  {
    echo("Routine: %s" % e.routine);
    echo("Number of effective arguments: %s" % e.effective);
  };
}
[00000132] *** Exception type: Arity
[00000133] *** Routine: cos
[00000134] *** Number of effective arguments: 2
```

urbiscript
Session

23.8.4 Finally

Using the finally-clause construct, you can ensure some code is executed upon exiting a try-clause, be it naturally or through an exception, `return`, `continue`, ...

23.8.4.1 Regular execution

The finally-clause is executed when the try-clause exits normally.

urbiscript
Session

```
try
{
  echo("inside");
}
finally
{
  echo("finally");
};
[00000001] *** inside
[00000002] *** finally
```

The value of the whole construct is that of the finally-clause.

urbiscript
Session

```
try { 51 } finally { 42 };
[00000001] 51
```

23.8.4.2 Control-flow

The finally clause is executed even if `return` is run.

urbiscript
Session

```
function with_return(var enable)
{
  try
  {
    echo("before return");
    if (enable)
      return;
    echo("after return");
  }
  finally
  {
    echo("finally");
  };
  echo("after try-block")
}

with_return(false);
[00001983] *** before return
[00001985] *** after return
[00001985] *** finally
[00001986] *** after try-block

with_return(true);
[00001991] *** before return
[00001992] *** finally
```

It is also the case when the control flow is disrupted by `continue` or `break`.

urbiscript
Session

```
for (var i : ["1", "continue", "2", "break", "3"])
  try
  {
    echo("before: " + i);
    switch (i)
    {
      case "break": break;
      case "continue": continue;
    };
    echo("after: " + i);
  }
  finally
  {
```

```

    echo("finally: " + i);
};

[00000663] *** before: 1
[00000671] *** after: 1
[00000671] *** finally: 1
[00000673] *** before: continue
[00000675] *** finally: continue
[00000682] *** before: 2
[00000703] *** after: 2
[00000703] *** finally: 2
[00000704] *** before: break
[00000705] *** finally: break

```

23.8.4.3 Exceptions

Exceptions caught in the try-catch clause are much like a regular execution flow. In particular, the value of the construct is that of the try-catch clause regardless of the execution of the `finally` clause.

```

try      { echo("try");      "try" }
catch (var e) { echo("catch");  "catch" }
finally   { echo("finally"); "finally" };
[00000614] *** try
[00000615] *** finally
[00000616] "try"

try      { echo("try");      "try" }
catch (var e) { echo("catch");  "catch" }
else      { echo("else");    "else" }
finally   { echo("finally"); "finally" };
[00000614] *** try
[00000615] *** else
[00000615] *** finally
[00000616] "else"

try                  { echo("throw 42"); throw 42; "try" }
catch (var e if e == 42) { echo("caught " + e);      "catch" }
finally               { echo("finally");           "finally" };
[00000626] *** throw 42
[00000626] *** caught 42
[00000631] *** finally
[00000631] "catch"

```

urbiscript
Session

Uncaught exceptions (i.e., exceptions for which there were no handlers) are propagated after the exception of the finally-clause.

```

try      { echo("throw"); throw 51; "try" }
catch (var e if e == 42) { echo("caught " + e);      "catch" }
finally   { echo("finally");           "finally" };
[00000616] *** throw
[00000617] *** finally
[00000625:error] !!! 51

```

urbiscript
Session

Exceptions launched in the finally-clause override previous exceptions.

```

try      { throw "throw" }
catch   { throw "catch" }
finally { throw "finally" };
[00005200:error] !!! finally

```

urbiscript
Session

23.9 Assertions

Assertions allow to embed consistency checks in the code. They are particularly useful when developing a program since they allow early catching of errors. Yet, they can be costly in production mode: the run-time cost of verifying every single assertion might be prohibitive. Therefore, as in C-like languages, assertions are disabled when `System NDEBUG` is true, see [System](#).

urbiscript supports assertions in two different ways: with a function-like syntax, which is adequate for single claims, and a block-like syntax, to group claims together.

23.9.1 Asserting an Expression

urbiscript
Session

```
assert(true);
assert(42);
```

Failed assertions are displayed in a user friendly fashion: first the assertion is displayed before evaluation, then the effective values are reported.

urbiscript
Session

```
function fail () { false }|;
assert (fail);
[00010239:error] !!! failed assertion: fail (fail == false)

function lazyFail { call.evalArgAt(0); false }|;
assert (lazyFail(1+2, "+* 2));
[00010241:error] !!! failed assertion: lazyFail(1.'+'(2), "+.*'(2)) (lazyFail(3, ?) == false)
```

The following example is more realistic.

urbiscript
Session

```
function areEqual
{
    var res = true;
    if (!call.args.empty)
    {
        var args = call.evalArgs;
        var a = args[0];
        for (var b : args.tail)
            if (a != b)
            {
                res = false;
                break;
            }
    };
    res
}|;
assert (areEqual);
assert (areEqual(1));
assert (areEqual(1, 0 + 1));
assert (areEqual(1, 1, 1+1));
[00001388:error] !!! failed assertion: areEqual(1, 1, 1.'+'(1)) (areEqual(1, 1, 2) == false)
assert (areEqual(1+2, 3+3, 4*6));
[00001393:error] !!! failed assertion: areEqual(1.'+'(2), 3.'+'(3), 4.*'(6)) (areEqual(3, 6, 24) == false)
```

Comparison operators are recognized, and displayed specially:

urbiscript
Session

```
assert(1 == 1 + 1);
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)
```

Note however that if opposite comparison operators are absurd (i.e., if for instance `a == b` is not true, but `a != b` is not true either), them the message is unlikely to make sense.

23.9.2 Assertion Blocks

Groups of assertions are more readable when used with the `assert{exp1; exp2; ...}` construct. The (possibly empty) list of claims may be ended with a semicolon.

```
assert
{
    true;
    42;
    1 == 1 + 1;
};

[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)
```

urbiscript
Session

For sake of readability and compactness, this documentation shows assertion blocks as follows.

```
true;
42;
1 == 1 + 1;
[00000002:error] !!! failed assertion: 1 == 1.'+'(1) (1 != 2)
```

Assertion
Block

23.10 Parallel and event-based flow control

23.10.1 at

Using the `at` construct, one can arm code that will be triggered each time some condition is true.

The `at` construct is as follows:

```
at (condition)
  statement1
onleave
  statement2
```

urbiscript
Session

The `condition` can be of two different kinds: `e?(args)` to catch when events are sent, or `exp` to catch each time a Boolean `exp` becomes true.

The `onleave statement2` part is optional. Note that, as is the case for the `if` statement, there must not be a semicolon after `statement1` if there is an `onleave` clause.

23.10.1.1 at on Events

See [Section 15.2](#) for an example of using `at` statements to watch events.

Durations Since events may last for a given duration (`e! ~ duration`), event handlers may also require an event to be sustained for a given amount of time before being “accepted” (`at (e? ~ duration)`).

```
var e = Event.new();
at (e?(var start) ~ 1s)
  echo("in : %s" % (time - start).round)
onleave
  echo("out: %s" % (time - start).round);

// This emission is too short to trigger the at.
e!(time);

// This one is long enough.
// The body triggers 1s after the emission started.
e!(time) ~ 2s;
[00001000] *** in : 1
[00002000] *** out: 2
```

urbiscript
Session

23.10.1.2 at on Boolean Expressions

The `at` construct can be used to watch a given Boolean expression.

```
urbiscript
Session
var x = 0 |
var x_is_two = false |
at (x == 2)
  x_is_two = true
onleave
  x_is_two = false;

x = 3|; assert(!x_is_two);
x = 2|; assert( x_is_two);
x = 2|; assert( x_is_two);
x = 3|; assert(!x_is_two);
```

It can also wait for some condition to hold long enough: $exp \sim duration$, as a condition, denotes the fact that exp was true for $duration$ seconds.

```
urbiscript
Session
var x = 0 |
var x_was_two_for_two_seconds = false |
at (x == 2 ~ 2s)
  x_was_two_for_two_seconds = true
onleave
  x_was_two_for_two_seconds = false;

x = 2      | assert(!x_was_two_for_two_seconds);
sleep(1.5s) | assert(!x_was_two_for_two_seconds);
sleep(1.5s) | assert( x_was_two_for_two_seconds);

x = 3|; sleep(0.1s); assert(!x_was_two_for_two_seconds);

x = 2      | assert(!x_was_two_for_two_seconds);
sleep(1.5s) | assert(!x_was_two_for_two_seconds);
x = 3|; x = 2|; sleep (1s) | assert(!x_was_two_for_two_seconds);
```

23.10.1.3 Synchronous and asynchronous at

By default, `at` is asynchronous: the enter and leave actions are executed in detached jobs and won't interfere with the execution flow of the job that triggered it.

```
urbiscript
Session
var e = Event.new;
[00000001] Event_OxADDR

at (e?)
{
  sleep(1s);
  echo("in");
}
onleave
{
  sleep(2s);
  echo("out");
};

e!;
// Actions are triggered in the background and won't block
// the execution flow.
sleep(500ms);
echo("Not blocked");
[00000002] *** Not blocked
sleep(1s);
[00000003] *** in
echo("Not blocked");
```

```
[00000004] *** Not blocked
sleep(500ms);
[00000003] *** out
```

When using the `sync` keyword after `at`, it becomes synchronous: when a job triggers it, all enter and leave actions are executed synchronously before the triggering statement returns.

```
var e = Event.new;
[00000001] Event_OxADDR

at sync (e?)
{
    sleep(1s);
    echo("in");
}
onleave
{
    sleep(1s);
    echo("out");
};

e!;
// Actions are triggered synchronously, the next line will be executed
// when they're done.
echo("Blocked");
[00000002] *** in
[00000003] *** out
[00000004] *** Blocked
```

urbiscript
Session

23.10.1.4 Scoping at `at`

`at` statements are not scoped. But, using a `Tag` object, one can control them. In the following example, `Tag.scope` is used to label the `at` statement. When the function ends, the `at` is no longer active.

```
var x = 0 |
var x_is_two = false |;

{
    Tag.scope:
        at (x == 2)
            x_is_two = true
        onleave
            x_is_two = false;
        sleep(2s);
},
x = 2 |; assert(x_is_two);
x = 1 |; assert(!x_is_two);
sleep(3s);
x = 2 | assert(!x_is_two);
```

urbiscript
Session

23.10.2 `every`

The `every` statement enables to execute a block of code repeatedly, with the given period.

```
// Print out a message every second.
timeout (2.1s)
every (1s)
    echo("Are you still there?");

[00000000] *** Are you still there?
[00001000] *** Are you still there?
[00002000] *** Are you still there?
```

urbiscript
Session

It exists in several flavors.

23.10.2.1 every|

The whole `every|` statement itself remains in foreground: statements attached after it with ; or | will not be reached unless you `break` out of it. You may use `continue` to finish one iteration. In that case, the following iteration is not immediately started, it will be launched as expected, at the given period.

urbiscript
Session

```
{
    var count = 4;
    var start = time;
    echo("before");
    every| (1s)
    {
        count -= 1;
        echo("begin: %s @ %1.0fs" % [count, time - start]);
        if (count == 2)
            continue;
        if (count == 0)
            break;
        echo("end: " + count);
    };
    echo("after");
};

[00000597] *** before
[00000598] *** begin: 3 @ 0s
[00000599] *** end: 3
[00000698] *** begin: 2 @ 1s
[00000798] *** begin: 1 @ 2s
[00000799] *** end: 1
[00000898] *** begin: 0 @ 3s
[00000899] *** after
```

The `every|` flavor does not let iterations overlap. If an iteration takes too long, the following iterations are delayed. That is, the next iterations will start immediately after the end of the current one, and next iterations will occur normally from this point.

urbiscript
Session

```
{
    var too_long = true|;

    var count = 5;
    // Every other iteration exceeds the period, and will delay the
    // following one.
    every| (1s)
    {
        if (! count --)
            break;

        if (too_long)
        {
            too_long = false;
            echo("Long in");
            sleep(1.5s);
            echo("Long out");
        }
        else
        {
            too_long = true;
            echo("Short");
        };
    };
};

[00000000] *** Long in
[00001500] *** Long out
[00001500] *** Short
[00002500] *** Long in
```

```
[00004000] *** Long out
[00004000] *** Short
```

The flow-control constructs `break` and `continue` are supported.

```
{
    var count = 0;
    every| (250ms)
    {
        count += 1;
        if (count == 2)
            continue;
        if (count == 4)
            break;
        echo(count);
    }
};

[00000000] *** 1
[00001500] *** 3
```

urbiscript
Session

23.10.3 watch

The `watch` construct is similar in spirit to using the `at` construct to monitor expressions, except it enables you to be notified when an arbitrary expression changed, not only when it becomes true or false. This makes `watch` a more primitive tool than `at` on expressions. Actually, `at` on expressions uses `watch` to determine when to reevaluate its condition.

`watch(expression)` evaluates to an `Event` that triggers every time `expression` changes, with its new value as payload.

```
var x = 0;
[00000000] 0
var y = 0;
[00000000] 0
var e = watch(x + y);
[00000000] Event_Ox103a1e978
at (e?(var value))
    echo("x + y = %s" % value);
x = 1;
[00000000] 1
[00000000] *** x + y = 1
y = 2;
[00000000] 2
[00000000] *** x + y = 3
```

urbiscript
Session

Note that “the expression changed” might be ambiguous: Urbi considers the expression to have changed when any component involved in its evaluation changed. If a `Float` is replaced with another `Float` of the same value, the expression has changed, since the new `Float` may have different slots.

```
var x = 0;
[00000000] 0
at (watch(x)?(var value))
    echo("x = %s" % value);
// This is considered as a change, although the new float value is also 0.
x = 0;
[00000000] 0
[00000000] *** x = 0
```

urbiscript
Session

Also, some modification may modify the evaluation, but still yield the same result.

```
var x = 1;
[00000000] 1
```

urbiscript
Session

```

at (watch(x % 2)?(var value))
  echo("x %% 2 = %s" % value);

// This is considered as a change, although the computation yields the
// same result.
x = 3;
[00000000] 3
[00000000] *** x % 2 = 1

```

23.10.3.1 every,

The default flavor, `every`, launches the execution of the block in the background every given period. Iterations may overlap.

urbiscript
Session

```

// If an iteration is longer than the given period, it will overlap
// with the next one.
timeout (2.8s)
  every (1s)
  {
    echo("In");
    sleep(1.5s);
    echo("Out");
  };
[00000000] *** In
[00001000] *** In
[00001500] *** Out
[00002000] *** In
[00002500] *** Out

```

23.10.4 for

The `for` loops come into several flavors, depending one the actual kind of `for` loop.

23.10.4.1 C-for,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

`for`, is syntactic sugar for `while`,, see [Section 23.10.8.1](#).

urbiscript
Session

```

for, (var i = 3; 0 < i; i -= 1)
{
  var j = i |
  echo ("in: i = %s, j = %s" % [i, j]);
  sleep(j/10);
  echo ("out: i = %s, j = %s" % [i, j]);
};
echo ("done");
[00000144] *** in: i = 3, j = 3
[00000145] *** in: i = 2, j = 2
[00000145] *** in: i = 1, j = 1
[00000246] *** out: i = 0, j = 1
[00000346] *** out: i = 0, j = 2
[00000445] *** out: i = 0, j = 3
[00000446] *** done

```

urbiscript
Session

```

for, (var i = 9; 0 < i; i -= 1)
{
  var j = i;
  if (j % 2)
    continue

```

```

else if (j == 4)
    break
else
    echo("%s: done" % j)
};

echo("done");
[00000146] *** 8: done
[00000148] *** 6: done
[00000150] *** done

```

23.10.4.2 range-for& (:)

One can iterate concurrently over the members of a collection.

```

for& (var i: [0, 1, 2])
{
    echo (i * i);
    echo (i * i);
};

[00000000] *** 0
[00000000] *** 1
[00000000] *** 4
[00000000] *** 0
[00000000] *** 1
[00000000] *** 4

```

urbiscript
Session

If an iteration executes `continue`, it is stopped; the other iterations are not affected.

```

for& (var i: [0, 1, 2])
{
    var j = i;
    if (j == 1)
        continue;
    echo (j);
};

[00020653] *** 0
[00021054] *** 2

```

urbiscript
Session

If an iteration executes `break`, all the iterations including this one, are stopped.

```

for& (var i: [0, 1, 2])
{
    var j = i;
    echo (j);
    if (j == 1)
        { echo ("break");
        break;};
    sleep(1s);
    echo (j);
};

[00000001] *** 0
[00000001] *** 1
[00000001] *** 2
[00000002] *** break

```

urbiscript
Session

23.10.4.3 for& (n)

Since `for& (n) body` is processed as `for& (var tmp: n) body`, which `tmp` a hidden variable, see Section 23.10.4.2 for details.

23.10.5 loop,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

This is syntactic sugar for `while, (true)`. In the following example, care must be taken that concurrent executions don't modify `n` simultaneously. This would happen had `;` been used instead of `|`.

Assertion Block

```
{
  var n = 10|;
  var res = []|;
  loop,
  {
    n-- |
    res << n |
    if (n == 0)
      break
  };
  res.sort
}
===[0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

23.10.6 waituntil

The `waituntil` construct is used to hold the execution until some condition is verified. Similarly to `at` (Section 23.10.1) and the other event-based constructs, `waituntil` may work on events, or on Boolean expressions.

23.10.6.1 waituntil on Events

When the execution flow enters a `waituntil`, the execution flow is held until the event is fired. Once caught, the event is consumed, another `waituntil` will require another event emission.

urbiscript Session

```
{
  var e = Event.new;
  {
    waituntil (e?);
    echo ("caught e");
  },
  e!;
  [00021054] *** caught e
  e!;
}
```

In the case of lasting events (see `Event.trigger`), the condition remains verified as long as the event is “on”.

urbiscript Session

```
{
  var e = Event.new;
  e.trigger;
  {
    waituntil (e?);
    echo ("caught e");
  };
  [00021054] *** caught e
  {
    waituntil (e?);
    echo ("caught e");
  };
  [00021054] *** caught e
```

```
{
  waituntil (e?);
  echo ("caught e");
};

[00021054] *** caught e
};
```

The event specification may use pattern-matching to specify the accepted events.

```
{
  var e = Event.new;
  {
    waituntil (e?(1, var b));
    echo ("caught e(1, %s)" % b);
  },
  e!;
  e!(1);
  e!(2, 2);
  e!(1, 2);
};

[00021054] *** caught e(1, 2)
e!(1, 2);
};
```

urbiscript
Session

Events sent before do not release the construct.

```
{
  var e = Event.new;
  e!;
  {
    waituntil (e?);
    echo ("caught e");
  },
  e!;
};

[00021054] *** caught e
};
```

urbiscript
Session

23.10.6.2 `waituntil` on Boolean Expressions

You may use any expression that evaluates to a truth value as argument to `waituntil`.

```
{
  var foo = Object.new;
  {
    waituntil (foo.hasLocalSlot("bar"));
    echo(foo.getLocalSlot("bar"));
  },
  var foo.bar = 123|;
};

[00021054] *** 123
```

urbiscript
Session

23.10.7 `whenever`

The `whenever` construct really behaves like a never-ending `loop if` construct. It also works on events and Boolean expressions, and triggers each time the condition *becomes* verified.

```
whenever (condition)
  statement1
```

urbiscript
Session

It supports an optional `else` clause, which is run whenever the condition changes “from true to false”.

```
whenever (condition)
  statement1
else
```

urbiscript
Session

statement2

The execution of a `whenever` clause is “instantaneous”, there is no mean to use ‘,’ to put it in background. It is also asynchronous with respect to the condition: the emission of an event is not held until all its watchers have completed their job.

23.10.7.1 whenever on Events

A `whenever` clause can be used to catch events with or without payloads.

urbiscript
Session

```
var e = Event.new();
whenever (e?)
    echo("e on")
else
    echo("e off");
[00000001] *** e off
[00000002] *** e off
[00000003] *** ...
e!;
[00000004] *** e on
[00000005] *** e off
[00000006] *** e off
[00000007] *** ...
e!(1) & e!(2);
[00000008] *** e on
[00000009] *** e on
[00000010] *** e off
[00000011] *** e off
[00000012] *** ...
```

The pattern-matching and guard on the payload is available.

urbiscript
Session

```
var e = Event.new();
whenever (e?("arg", var arg) if arg % 2)
    echo("e (%s) on" % arg)
else
    echo("e off");
e!("param", 23);
e!("arg", 52);
e!("arg", 23);
[00000001] *** e (23) on
[00000002] *** e off
[00000003] *** e off
[00000004] *** ...
e!("arg", 52);
e!("arg", 17);
[00000005] *** e (17) on
[00000006] *** e off
[00000007] *** e off
[00000008] *** ...
```

If the body of the `whenever` lasts for a long time, it is possible that two executions be run concurrently.

urbiscript
Session

```
var e = Event.new();
whenever (e?(var d))
{
    echo("e (%s) on begin" % d);
    sleep(d);
    echo("e (%s) on end" % d);
};

e!(0.3s) & e!(1s);
sleep(3s);
[00000202] *** e (1) on begin
```

```
[00000202] *** e (0.3) on begin
[00000508] *** e (0.3) on end
[00001208] *** e (1) on end
```

23.10.7.2 whenever on Boolean Expressions

A `whenever` construct will repeatedly evaluate its body as long as its condition holds. The number of evaluation of the bodies is typically non-deterministic, as not only does it depend on how long the condition holds, but also “how fast” the Urbi kernel runs.

```
urbiscript
Session

var x = 0|;
var count = 0|;
var t = Tag.new();
t:
    whenever (x % 2)
    {
        if (!count)
            echo("x is now odd (%s)" % x);
        count++;
    }
    else
    {
        if (!count)
            echo("x is now even (%s)" % x);
        count++;
    };
t:
    whenever (100 < count)
    {
        count = 0 |
        x++;
    };
waituntil(x == 4);
[00000769] *** x is now even (0)
[00000809] *** x is now odd (1)
[00000846] *** x is now even (2)
[00000886] *** x is now odd (3)
[00000924] *** x is now even (4)
t.stop;
```

23.10.8 While

23.10.8.1 while,

This feature is experimental. It might be changed, or even removed. Feedback on its use would be appreciated.

This construct provides a means to run concurrently multiple instances of statements. The semantics of

```
while, (condition)
    body;
```

urbiscript
Session

is the same as

```
condition | body , condition | body , ...
```

urbiscript
Session

Attention must be paid to the fact that the (concurrent) iterations share a common access to the environment, therefore if, for instance, you want to keep the value of some index variable, use a local variable inside the loop body:

urbiscript
Session

```
{
  var i = 4|
  while, (i)
  {
    var j = i -= 1;
    echo ("in: i = %s, j = %s" % [i, j]);
    sleep(j/10);
    echo ("out: i = %s, j = %s" % [i, j]);
  }|
  echo ("done");
}|
[00000144] *** in: i = 2, j = 3
[00000145] *** in: i = 1, j = 2
[00000145] *** in: i = 0, j = 1
[00000146] *** in: i = 0, j = 0
[00000146] *** out: i = 0, j = 0
[00000246] *** out: i = 0, j = 1
[00000346] *** out: i = 0, j = 2
[00000445] *** out: i = 0, j = 3
[00000446] *** done
```

As for the other flavors, `continue` skips the current iteration, and `break` ends the loop. Note that `break` stops all the running iterations. This semantics is likely to be changed to “`break` ends the current iteration and stops the generation of others, but lets the other concurrent iterations finish”, so do not rely on this feature.

Control flow is passed to the following statement when all the iterations are done.

urbiscript
Session

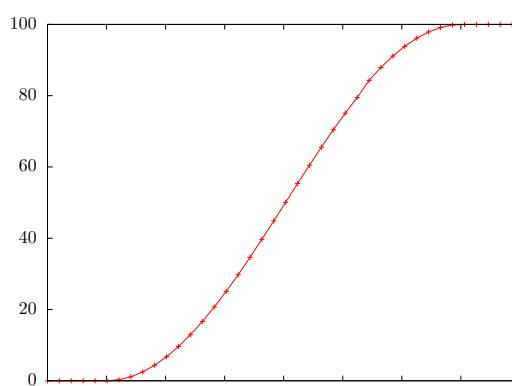
```
{
  var i = 10|
  while, (i)
  {
    var j = i -= 1;
    if (j % 2)
      continue
    else if (j == 4)
      break
    else
      echo("%s: done" % j);
  }|
  echo("done");
};|
[00000146] *** 8: done
[00000148] *** 6: done
[00000150] *** done
```

23.11 Trajectories

In robotics, *trajectories* are often used: they are a means to change the value of a variable (actually, a slot) over time. This can be done using detached executions, for instance using a combination of `every` and `detach`, but urbiscript provides syntactic sugar to this end.

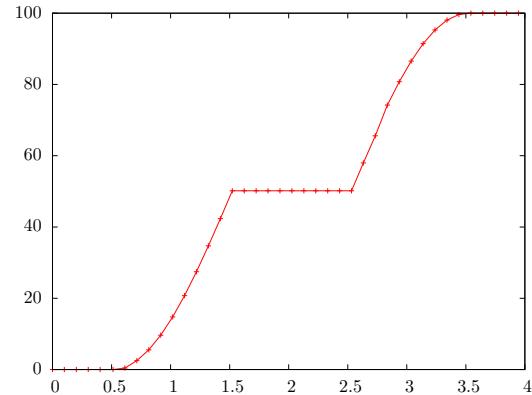
For instance the following drawing shows how the `y` variable is moved smoothly from its *initial value* (0) to its *target value* (100) in 3 seconds (the value given to the `smooth` attribute).

```
var y = 0;
{
  sleep(0.5s);
  y = 100 smooth:3s,
},
```



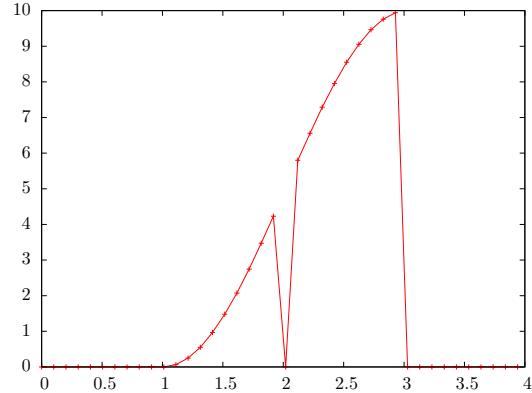
Trajectories can be frozen and unfrozen, using tags (Section 14.3). In that case, “time is suspended”, and the trajectory resumes as if the trajectory was never interrupted.

```
var y = 0;
{
    sleep(0.5s);
    assign: y = 100 smooth:2s,
    sleep(1s);
    assign.freeze;
    sleep(1s);
    assign.unfreeze;
},
```



When the target value is reached, the trajectory generator is detached from the variables: changes to the value of the variable no longer trigger the trajectory generator.

```
var y = 0;
{
    sleep(1s);
    assign: y = 10 smooth:2s,
    sleep(1s);
    y = 0;
    sleep(1s);
    y = 0;
},
```



See the specifications of [TrajectoryGenerator](#) for the list of supported trajectories.

23.12 Garbage collection and limitations

urbiscript provides automatic garbage collection. That is, you can create new objects and don’t have to worry about reclaiming the memory when you’re done with them. We use a reference counting algorithm for garbage collection: every object has a counter indicating how many references to it exist. When that counter drops to zero, nobody has a reference to the object anymore, and it is thus deleted.

```
{
    var x = List.new; // A new list is allocated
    x << 42;
};

// The list will be automatically freed, since there are no references to it left.
```

urbiscript
Session

This is not part of the language interface, and we might change the garbage collecting system in the future. Therefore, do not rely on the current garbage collecting behavior, and especially not on the determinism of the destruction time of objects.

However, this implementation has a limitation you should be aware of: cycle of object references won’t be properly reclaimed. Indeed if object A has a reference to B, and B has a reference to A, none of them will ever be reclaimed since they both have a reference pointing to them. As a consequence, avoid creating cycles in object references, or if you really have to, break the cycle manually before releasing your last reference to the object.

urbiscript
Session

```
// Create a reference cycle
var A = Object.new;
var A.B = Object.new; // A refers to B
var A.B.A = A; // B refers back to A

removeLocalSlot("A"); // delete our last reference to A
// Although we have no reference left to A or B,
// they won't be deleted since they refer to each other.
```

If you really need the cycle, this is how you could break it manually:

urbiscript
Session

```
A.B.removeLocalSlot("A"); // Break the cycle
removeLocalSlot("A"); // Delete our last reference to A
// A will be deleted since it's not referred from anywhere.
// Since A held the last reference to B, B will be deleted too.
```

Chapter 24

urbiscript Standard Library

24.1 Barrier

`Barrier` is used to wait until another job raises a signal. This can be used to implement blocking calls waiting until a resource is available.

24.1.1 Prototypes

- `Object`

24.1.2 Construction

A `Barrier` can be created with no argument. Calls to `signal` and `wait` done on this instance are restricted to this instance.

```
Barrier.new;
[00000000] Barrier_0x25d2280
```

urbiscript
Session

24.1.3 Slots

- `signal(payload)`

Wake up one of the job waiting for a signal. The `payload` is sent to the `wait` method. Return the number of jobs woken up.

```
do (Barrier.new)
{
    echo(wait) &
    echo(wait) &
    assert
    {
        signal(1) == 1;
        signal(2) == 1;
    }
}|;
[00000000] *** 1
[00000000] *** 2
```

urbiscript
Session

- `signalAll(payload)`

Wake up all the jobs waiting for a signal. The `payload` is sent to all `wait` methods. Return the number of jobs woken up.

```
do (Barrier.new)
{
    echo(wait) &
    echo(wait) &
    assert
```

urbiscript
Session

```
{
    signalAll(1) == 2;
    signalAll(2) == 0;
}
};

[00000000] *** 1
[00000000] *** 1
```

- **wait**

Block until a signal is received. The *payload* sent with the signal function is returned by the `wait` method.

urbiscript
Session

```
do (Barrier.new)
{
    echo(wait) &
    signal(1)
};
[00000000] *** 1
```

24.2 Binary

A Binary object, sometimes called a *blob*, is raw memory, decorated with a user defined header.

24.2.1 Prototypes

- **Object**

24.2.2 Construction

Binaries are usually not made by users, but they are heavily used by the internal machinery when exchanging Binary UVValues. A binary features some **content** and some **keywords**, both simple **Strings**.

urbiscript
Session

```
Binary.new("my header", "my content");
[00000001] BIN 10 my header
my content
```

Beware that the third line above ('`my content`'), was output by the system, although not preceded by a timestamp.

24.2.3 Slots

- **'+'** (*that*)

Return a new Binary whose keywords are those of `this` if not empty, otherwise those of `that`, and whose data is the concatenation of both.

Assertion
Block

```
Binary.new("0", "0") + Binary.new("1", "1")
== Binary.new("0", "01");
Binary.new("", "0") + Binary.new("1", "1")
== Binary.new("1", "01");
```

- **'=='** (*other*)

Whether **keywords** and **data** are equal.

Assertion
Block

```
Binary.new("0", "0") == Binary.new("0", "0");
Binary.new("0", "0") != Binary.new("0", "1");
Binary.new("0", "0") != Binary.new("1", "0");
```

- **asString**

Display using the syntactic rules of the UObject/UValue protocol. Incoming binaries must use a semicolon to separate the header part from the content, while outgoing binaries use a carriage-return.

```
assert(Binary.new("head", "content").asString
      == "BIN 7 head\ncontent");
var b = BIN 7 header;content;
[00000002] BIN 7 header
content
assert(b == Binary.new("header", "content"));
```

urbiscript
Session

This syntax (`BIN size header; content`) is *partially* supported in urbiscript, but it is strongly discouraged. Rather, use the `\B(size)(data)` special escape (see [Section 23.1.6.6](#)):

```
Binary.new("head", "\B(7)(content)").asString
      == "BIN 7 head\ncontent";
```

Assertion
Block

- **data**

The data carried by the Binary.

```
Binary.new("head", "content").data == "content";
```

Assertion
Block

- **empty**

Whether the data is empty.

```
Binary.new("head", "").empty;
!Binary.new("head", "content").empty;
```

Assertion
Block

- **keywords**

The headers carried by the Binary.

```
Binary.new("head", "content").keywords == "head";
```

Assertion
Block

24.3 Boolean

There is no object `Boolean` in urbiscript, but two specific objects `true` and `false`. They are the result of all the comparison statements.

24.3.1 Prototypes

The objects `true` and `false` have the following prototype.

- **Singleton**

24.3.2 Construction

There are no constructors, use `true` and `false`. Since they are singletons, `clone` will return themselves, not new copies.

```
true;
!false;
2 < 6 === true;
true.new === true;
6 < 2 === false;
```

Assertion
Block

24.3.3 Truth Values

As in many programming languages, conditions may be more than only `true` and `false`. Whether some value is considered as true depends on the type of `this`. Actually, by default objects as considered “true”, objects evaluating to “false” are the exception:

- `false, nil`
`false.`

`void` raise an error.

`Float false iff null (Float).`

- `Dictionary, List, String`
`false iff empty (Dictionary, List, String).`
- otherwise
`true.`

The method `Object.asBool` is in charge of converting some arbitrary value into a Boolean.

urbiscript
Session

```
assert(Global.asBool == true);
assert(nil.asBool == false);
void.asBool;
[00000421:error] !!! unexpected void
```

24.3.4 Slots

- `!`

Logical negation. If `this` is `false` return `true` and vice-versa.

Assertion
Block

```
!true == false;
!false == true;
```

- `asBool`

Identity.

Assertion
Block

```
true.asBool == true;
false.asBool == false;
```

24.4 CallMessage

Capturing a method invocation: its target and arguments.

24.4.1 Examples

24.4.1.1 Evaluating an argument several times

The following example implements a lazy function which takes an integer `n`, then arguments. The `n`-th argument is evaluated twice using `evalArgAt`.

urbiscript
Session

```
function callTwice
{
  var n = call.evalArgAt(0);
  call.evalArgAt(n);
  call.evalArgAt(n)
} |;

// Call twice echo("foo").
```

```
callTwice(1, echo("foo"), echo("bar"));
[00000001] *** foo
[00000002] *** foo

// Call twice echo("bar").
callTwice(2, echo("foo"), echo("bar"));
[00000003] *** bar
[00000004] *** bar
```

24.4.1.2 Strict Functions

Strict functions do support `call`.

```
function strict(x)
{
    echo("Entering");
    echo("Strict: " + x);
    echo("Lazy: " + call.evalArgAt(0));
} |;

strict({echo("1"); 1});
[00000011] *** 1
[00000013] *** Entering
[00000012] *** Strict: 1
[00000013] *** 1
[00000014] *** Lazy: 1
```

urbiscript
Session

24.4.2 Slots

- `args`

The list of not yet evaluated arguments.

```
function args { call.args }|
assert
{
    args == [];
    args() == [];
    args({echo(111); 1}) == [Lazy.new(closure() {echo(111); 1})];
    args(1, 2) == [Lazy.new(closure () {1}),
                    Lazy.new(closure () {2})];
};
```

urbiscript
Session

- `argsCount`

The number of arguments. Do not evaluate them.

```
function argsCount { call.argsCount }|
assert
{
    argsCount == 0;
    argsCount() == 0;
    argsCount({echo(1); 1}) == 1;
    argsCount({echo(1); 1}, {echo(2); 2}) == 2;
};
```

urbiscript
Session

- `code`

The body of the called function as a `Code`.

```
function code { call.getSlot("code") }|
assert (code == getSlot("code"));
```

urbiscript
Session

- **eval**

Evaluate `this`, and return the result.

urbiscript
Session

```
var c1 = do (CallMessage.new)
{
    var this.target = 1;
    var this.message = "+";
    var this.args = [Lazy.new(function () {2})];
}
assert { c1.eval == 3 };

// A lazy function that returns the sum of this and the second argument,
// regardless of the first argument.
function Float.addSecond
{
    this + call.evalArgAt(1);
}
var c2 = do (CallMessage.new)
{
    var this.target = 2;
    var this.message = "addSecond";
    var this.args = [Lazy.new(function () { assert (false) }),
                    Lazy.new(function () { echo (5); 5 })];
}
assert { c2.eval == 7 };
[00000454] *** 5
```

- **evalArgAt(*n*)**

Evaluate the *n*-th argument, and return its value. *n* must evaluate to an non-negative integer. Repeated invocations repeat the evaluation, see [Section 24.4.1.1](#).

urbiscript
Session

```
function sumTwice
{
    var n = call.evalArgAt(0);
    call.evalArgAt(n) + call.evalArgAt(n)
};

function one () { echo("one"); 1 }();

sumTwice(1, one, one + one);
[00000008] *** one
[00000009] *** one
[00000010] 2
sumTwice(2, one, one + one);
[00000011] *** one
[00000012] *** one
[00000013] *** one
[00000014] *** one
[00000015] *** one
[00000016] 4

sumTwice(3, one, one);
[00000017:error] !!! evalArgAt: invalid index: 3
[00000018:error] !!!     called from: sumTwice
sumTwice(3.14, one, one);
[00000019:error] !!! evalArgAt: invalid index: 3.14
[00000020:error] !!!     called from: sumTwice
```

- **evalArgs**

Call `evalArgAt` for each argument, return the list of values.

urbiscript
Session

```
function twice
{
    call.evalArgs + call.evalArgs
};
```

```
twice({echo(1); 1}, {echo(2); 2});
[00000011] *** 1
[00000012] *** 2
[00000011] *** 1
[00000012] *** 2
[00000013] [1, 2, 1, 2]
```

- **message**

The name under which the function was called.

```
function myself { call.message }|
assert(myself == "myself");
```

urbiscript
Session

- **sender**

The object *from which* the invocation was made (the *caller* in other languages). Not to be confused with **target**.

```
function Global.getSender { call.sender } |
function Global.callGetSender { getSender } |

assert
{
    // Call from the current Lobby, with the Lobby as target.
    getSender === lobby;
    // Call from the current Lobby, with Global as the target.
    Global.getSender === lobby;
    // Ask Lobby to call getSender.
    callGetSender === lobby;
    // Ask Global to call getSender.
    Global.callGetSender === Global;
};
```

urbiscript
Session

- **target**

The object *on which* the invocation is made. In other words, the object that will be bound to **this** during the evaluation. Not to be confused with **sender**.

```
function Global.getTarget { call.target } |
function Global.callGetTarget { getTarget } |

assert
{
    // Call from the current Lobby, with the Lobby as target.
    getTarget === lobby;
    // Call from the current Lobby, with Global as the target.
    Global.getTarget === Global;
    // Ask Lobby to call getTarget.
    callGetTarget === lobby;
    // Ask Global to call getTarget.
    Global.callGetTarget === Global;
};
```

urbiscript
Session

24.5 Channel

Returning data, typically asynchronous, with a label so that the “caller” can find it in the flow.

24.5.1 Prototypes

- **Object**

24.5.2 Construction

Channels are created like any other object. The constructor must be called with a string which will be the label.

```
urbiscript
Session
var ch1 = Channel.new("my_label");
[00000201] Channel_0x7985810

ch1 << 1;
[00000201:my_label] 1

var ch2 = ch1;
[00000201] Channel_0x7985810

ch2 << 1/2;
[00000201:my_label] 0.5
```

24.5.3 Slots

- `'<<'(value)`

Send `value` to `this` tagged by its label if non-empty.

```
urbiscript
Session
Channel.new("label") << 42;
[00000000:label] 42

Channel.new("") << 51;
[00000000] 51
```

- `echo(value)`

Same as `lobby.echo(value, name)`, see [Lobby.echo](#).

```
urbiscript
Session
Channel.new("label").echo(42);
[00000000:label] *** 42

Channel.new("").echo("Foo");
[00000000] *** Foo
```

- `enabled`

Whether the Channel is enabled. Disabled Channels produce no output.

```
urbiscript
Session
var c = Channel.new("")|;

c << "enabled";
[00000000] "enabled"

c.enabled = false|;
c << "disabled";

c.enabled = true|;
c << "enabled";
[00000000] "enabled"
```

- `Filter`

Filtering channel.

The Filter channel outputs text that can be parsed without error by the liburbi. It does this by filtering types not handled by the liburbi, and displaying them using `echo`.

```
urbiscript
Session
// Use a filtering channel on our lobby output.
topLevel = Channel.Filter.new("")|;
// liburbi knows about List, Dictionary, String and Float, so standard display.
[1, "foo", ["test" => 5]];
```

```
[00000001] [1, "foo", ["test" => 5]]
// liburbi does not know 'lobby', so it is escaped with echo:
lobby;
[00000002] *** Lobby_0xADDR
// The following list contains a function which is not handled by liburbi, so
// it gets escaped too.
[1, function () {}];
[00000003] *** [1, function () {}]
// Restore default display to see the difference.
topLevel = Channel.topLevel;
// The echo is now gone.
[1, function () {}];
[00001758] [1, function () {}]
```

- **quote**

Whether the strings are output escaped (the default) instead of raw strings.

```
var d = Channel.new("")|;

assert(d.enabled);
d << "A \"String\"";
[00000000] "A \"String\""

d.quote = false;
d << "A \"String\"";
[00000000] A "String"
```

urbiscript
Session

- **name**

The name of the Channel, used to label the output.

```
assert
{
    Channel.new("").name == "";
    Channel.new("foo").name == "foo";
};
```

urbiscript
Session

- **null**

A predefined stream whose `enabled` is `false`.

```
Channel.null << "Message";
```

urbiscript
Session

- **topLevel**

A predefined stream for regular output. Strings are output escaped.

```
Channel.topLevel << "Message";
[00015895] "Message"
Channel.topLevel << "\"quote\"";
[00015895] "\"quote\""
```

urbiscript
Session

- **warning**

A predefined stream for warning messages. Strings sent to it are not escaped.

```
Channel.warning << "Message";
[00015895:warning] Message
Channel.warning << "\"quote\"";
[00015895] "quote"
```

urbiscript
Session

24.6 Code

Functions written in urbiscript.

24.6.1 Prototypes

- Comparable
- Executable

24.6.2 Construction

The keywords `function` and `closure` build Code instances.

Assertion Block

```
function(){}.protos[0] === &Code;
closure (){}.protos[0] === &Code;
```

24.6.3 Slots

- Whether `this` and `that` are the same source code (actually checks that both have the same `asString`), and same closed values.

Closures and functions are different, even if the body is the same.

Assertion Block

```
function () { 1 } == function () { 1 };
function () { 1 } != closure () { 1 };
closure () { 1 } != function () { 1 };
closure () { 1 } == closure () { 1 };
```

No form of equivalence is applied on the body, it must be the same.

Assertion Block

```
function () { 1 + 1 } == function () { 1 + 1 };
function () { 1 + 2 } != function () { 2 + 1 };
```

Arguments do matter, even if in practice the functions are the same.

Assertion Block

```
function (var ignored) {} != function () {};
function (var x) { x } != function (y) { y };
```

A lazy function cannot be equal to a strict one.

Assertion Block

```
function () { 1 } != function { 1 };
```

If the functions capture different variables, they are different.

urbiscript Session

```
{
  var x;
  function Global.capture_x() { x };
  function Global.capture_x_again () { x };
  {
    var x;
    function Global.capture_another_x() { x };
  }
};

assert
{
  &capture_x == &capture_x_again;
  &capture_x != &capture_another_x;
};
```

If the functions capture different targets, they are different.

urbiscript Session

```
class Foo
{
  function makeFunction() { function () {} };
  function makeClosure() { closure () {} };
};
```

```

class Bar
{
    function makeFunction() { function () {} };
    function makeClosure() { closure () {} };
};

assert
{
    Foo.makeFunction() == Bar.makeFunction();
    Foo.makeClosure() != Bar.makeClosure();
};

```

- **apply(*args*)**

Invoke the routine, with all the arguments. The target, `this`, will be set to `args[0]` and the remaining arguments will be given as arguments.

```

function (x, y) { x+y }.apply([nil, 10, 20]) == 30;
function () { this }.apply([123]) == 123;

// There is Object.apply.
1.apply([this]) == 1;

```

Assertion Block

```

function () {}.apply([]);
[00000001:error] !!! apply: argument list must begin with 'this'

function () {}.apply([1, 2]);
[00000002:error] !!! apply: expected 0 argument, given 1

```

urbiscript Session

- **asString**

Conversion to `String`.

```

closure () { 1 }.asString == "closure () { 1 }";
function () { 1 }.asString == "function () { 1 }";

```

Assertion Block

- **bodyString**

Conversion to `String` of the routine body.

```

closure () { 1 }.bodyString == "1";
function () { 1 }.bodyString == "1";

```

Assertion Block

24.7 Comparable

Objects that can be compared for equality and inequality. See also [Orderable](#).

This object, made to serve as prototype, provides a definition of `!=` based on `==`. `Object` provides a default implementation of `==` that bounces on the physical equality `====`.

```

class Foo : Comparable
{
    var value = 0;
    function init (v) { value = v; };
    function '==' (that) { value == that.value; };
};

assert
{
    Foo.new(1) == Foo.new(1);
    Foo.new(1) != Foo.new(2);
};

```

urbiscript Session

24.7.1 Slots

- Whether `! (this != that)`.

urbiscript
Session

```
class FortyTwo : Comparable
{
    function '!=' (that) { 42 != that };
}|;
assert
{
    FortyTwo != 51;
    FortyTwo == 42;
};
```

- `!= (\var{that})` Whether `! (this == that)`.

urbiscript
Session

```
class FiftyOne : Comparable
{
    function '==' (that) { 51 == that };
}|;
assert
{
    FiftyOne == 51;
    FiftyOne != 42;
};
```

24.8 Container

This object is meant to be used as a prototype for objects that support `has` and `hasNot` methods. Any class using this prototype must redefine either `has`, `hasNot` or both.

24.8.1 Prototypes

- `Object`

24.8.2 Slots

- `has(e)`

`!hasNot(e)`. The indented semantics is “true when the container has a key (or item) matching `e`”. This is what `e in c` is mapped onto.

urbiscript
Session

```
class NotCell : Container
{
    var val;
    function init(var v) { val = v };
    function hasNot(var v) { val != v };
}|;
var c = NotCell.new(23)|;
assert
{
    c.has(23);      23 in c;
    c.hasNot(3);   3 not in c;
};
```

- `hasNot(e)`

`!has(e)`. The indented semantics is “true when the container does not have a key (or item) matching `e`”.

urbiscript
Session

```

class Cell : Container
{
    var val;
    function init(var v) { val = v };
    function has(var v) { val == v };
}|;
var d = Cell.new(23)|;
assert
{
    d.has(23);      23 in d;
    d.hasNot(3);   3 not in d;
};

```

24.9 Control

`Control` is designed as a namespace for control sequences. Some of these entities are used by the Urbi engine to execute some urbiscript features; in other words, users are not expected to you use it, much less change it.

24.9.1 Prototypes

- `Object`

24.9.2 Slots

- `detach(exp)`

Detach the evaluation of the expression `exp` from the current evaluation. The `exp` is evaluated in parallel to the current code and keep the current tag which are attached to it. Return the spawned `Job`. Same as calling `System.spawn: System.spawn(closure () { exp })`, `true`.

```

{
    var jobs = [];
    var res = [];
    for (var i : [0, 1, 2])
    {
        jobs << detach({ res << i; res << i });
        if (i == 2)
            break;
    };
    assert (res == [0, 1, 0]);
    jobs
};

[00009120] [Job<shell_11>, Job<shell_12>, Job<shell_13>]

```

urbiscript
Session

- `disown(exp)`

Same as `detach` except that tags used to tag the `disown` call are not inherited inside the expression. Return the spawned `Job`. Same as calling `System.spawn: System.spawn(closure () { exp })`,

```

{
    var jobs = [];
    var res = [];
    for (var i : [0, 1, 2])
    {
        jobs << disown({ res << i; res << i });
        if (i == 2)
            break;
    };
    jobs.each (function (var j) { j.waitForTermination });
    assert (res == [0, 1, 0, 2, 1, 2]);
};


```

urbiscript
Session

```
    jobs
};

[00009120] [Job<shell_14>, Job<shell_15>, Job<shell_16>]
```

- **`persist(expression, delay)`**

Return an object whose `val` slot evaluates to true if the `expression` has been continuously true for this `delay` and false otherwise.

This function is used to implement

urbiscript
Session

```
at (condition ~ delay)
action
```

as

urbiscript
Session

```
var u = persist (condition, delay);
at (u.val)
action
```

The `persist` action will be controlled by the same tags as the initial `at` block.

24.10 Date

This class is meant to record dates in time, with microsecond resolution.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

24.10.1 Prototypes

- [Orderable](#)
- [Comparable](#)

24.10.2 Construction

Without argument, newly constructed Dates refer to the current date.

urbiscript
Session

```
Date.new;
[00000001] 2010-08-17 14:40:52.549726
```

With a string argument `d`, refers to the date contained in `d`. The string should be formatted as ‘`yyyy-mm-dd hh:mm:ss`’ (see [asString](#)). `mm` and `ss` are optional. If the block ‘`hh:mm:ss`’ is absent, the behavior is undefined.

urbiscript
Session

```
Date.new("2003-10-10 20:10:50:637");
[00000001] 2003-10-10 20:10:50.637000

Date.new("2003-10-10 20:10:50");
[00000001] 2003-10-10 20:10:50.000000

Date.new("2003-Oct-10 20:10");
[00000002] 2003-10-10 20:10:00.000000

Date.new("2003-10-10 20");
[00000003] 2003-10-10 20:00:00.000000
```

Pay attention that the format is rather strict; for instance too many spaces between day and time result in an error.

urbiscript
Session

```
Date.new("2003-10-10 20:10:50");
[00001968:error] !!! new: cannot convert to date: 2003-10-10 20:10:50
```

Pay attention that the format is not strict enough either; for instance, below, the ‘.’ separator seem to prefix microseconds, but actually merely denotes the minutes. Seconds must be spelled out in order to introduce microseconds.

```
Date.new("2003-10-10 00.12");
[00000003] 2003-10-10 00:12:00.000000

Date.new("2003-10-10 00:00.12");
[00000003] 2003-10-10 00:00:12.000000
```

urbiscript
Session

24.10.3 Slots

- `'+'(that)`

The date which corresponds to waiting `Duration that` after `this`.

```
Date.new("2010-08-17 12:00:00.2") + 63.2s == Date.new("2010-08-17 12:01:03.4");
```

Assertion
Block

- `'-'(that)`

If `that` is a Date, the difference between `this` and `that` as a `Duration`.

```
Date.new("2010-08-17 12:01:00.50") - Date.new("2010-08-17 12:00") == 60.5s;
Date.new("2010-08-17 12:00") - Date.new("2010-08-17 12:01") == -60s;
```

Assertion
Block

If `that` is a Duration or a Float, the corresponding Date.

```
Date.new("2010-08-17 12:01") - 60s == Date.new("2010-08-17 12:00");
Date.new("2010-08-17 12:01") - 60s
== Date.new("2010-08-17 12:01") - Duration.new(60s);
```

Assertion
Block

- `'=='(that)`

Equality test.

```
Date.new("2010-08-17 12:00:00.123") == Date.new("2010-08-17 12:00:00.123");
Date.new("2010-08-17 12:00") != Date.new("2010-08-17 12:01");
```

Assertion
Block

- `'<'(that)`

Order comparison.

```
Date.new("2010-08-17 12:00") < Date.new("2010-08-17 12:01");
!(Date.new("2010-08-17 12:01") < Date.new("2010-08-17 12:00"));
```

Assertion
Block

- `asFloat`

The duration since the `epoch`, as a Float.

```
var d = Date.new("2002-01-20 23:59:59")|;
assert
{
    d.asFloat == d - d.epoch;
    d.asFloat.isA(Float);
};
```

urbiscript
Session

- `asString`

Present as ‘`yyyy-mm-dd hh:mm:ss.us`’ where:

- `yyyy` is the four-digit year,
- `mm` the three letters name of the month (Jan, Feb, ...),
- `dd` the two-digit day in the month (from 1 to 31),
- `hh` the two-digit hour (from 0 to 23),

- *mn* the two-digit number of minutes in the hour (from 0 to 59),
- *ss* the two-digit number of seconds in the minute (from 0 to 59), and
- *iiiiii* the six-digit number of microseconds.

Assertion Block

```
Date.new("2009-02-14 00:31:30").asString == "2009-02-14 00:31:30.000000";
```

- **day**

The day as a [Float](#).

urbiscript Session

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.day == 29);
  d.day = 1;
  assert(d == Date.new("2010-09-01 17:32:53"));
};
```

urbiscript Session

```
Date.new("2010-02-01 17:32:53").day = 29;
[00000001:error] !!! updateSlot: Day of month is not valid for year
```

- **epoch**

A fixed value, the “origin of times”: January 1st 1970, at midnight.

urbiscript Session

```
Date.epoch == Date.new("1970-01-01 00:00:00.00");
```

- **hour**

The hour as a [Float](#). Always less than 24.

urbiscript Session

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.hour == 17);
  d.hour = 8;
  assert(d == Date.new("2010-09-29 08:32:53"));
};
```

- **microsecond**

The number of microseconds in the current second, as a [Float](#). See also [us](#). Always less than 1000000.

urbiscript Session

```
{
  var d = Date.new("2010-09-29 17:32:53.123456");
  assert(d.microsecond == 123456);
  d.microsecond = 654321;
  assert(d == Date.new("2010-09-29 17:32:53.654321"));
};
```

- **minute**

The minute as a [Float](#). Always less than 60.

urbiscript Session

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.minute == 32);
  d.minute = 12;
  assert(d == Date.new("2010-09-29 17:12:53"));
};
```

- **month**

The month as a [Float](#). Always less or equal to 12.

urbiscript Session

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.month == 9);
  d.month = 3;
  assert(d == Date.new("2010-03-29 17:32:53"));
};
```

- **now**

The current date. Equivalent to Date.new.

```
Date.now;
[00000000] 2012-03-02 15:31:42
```

urbiscript
Session

- **second**

The second as a [Float](#).

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.second == 53);
  d.second = 37;
  assert(d == Date.new("2010-09-29 17:32:37"));
};
```

urbiscript
Session

- **timestamp**

Synonym for [asFloat](#).

- **us**

The *total* number of microseconds since midnight, as a [Float](#). See also [microsecond](#).

```
Date.new("2010-08-17 00:00:00.0") .us == 0;
Date.new("2010-08-17 00:00:00.123456") .us == 123456;
Date.new("2010-08-17 00:00:01.234567") .us == 1234567;
Date.new("2010-08-17 01:02:03.456789") .us
  == (1 * 3600 + 2 * 60 + 3) * 1000000 + 456789;
```

Assertion
Block

```
{
  var d = Date.new("2010-09-29 17:32:53.123456");
  assert(d.us == 63173123456);
  d.us = 123456;
  assert(d == Date.new("2010-09-29 00:00:00.123456"));
};
```

urbiscript
Session

- **year**

The year as a [Float](#).

```
{
  var d = Date.new("2010-09-29 17:32:53");
  assert(d.year == 2010);
  d.year = 2000;
  assert(d == Date.new("2000-09-29 17:32:53"));
};
```

urbiscript
Session

24.11 Dictionary

A *dictionary* is an *associative array*, also known as a *hash* in some programming languages. They are arrays whose indexes are arbitrary objects.

24.11.1 Example

The following session demonstrates the features of the Dictionary objects.

urbiscript
Session

```
var d = ["one" => 1, "two" => 2];
[00000001] ["one" => 1, "two" => 2]

for (var p : d)
    echo (p.first + " => " + p.second);
[00000003] *** one => 1
[00000002] *** two => 2



```

24.11.2 Hash values

Arbitrary objects can be used as dictionary keys. To map to the same cell, two objects used as keys must have equal hashes (retrieved with the `Object.hash` method) and be equal to each other (in the `Object.'=='` sense).

This means that two different objects may have the same hash: the equality operator (`Object.'=='`) is checked in addition to the hash, to handle such collision. However a good hash algorithm should avoid this case, since it hinders performances.

See `Object.hash` for more detail on how to override hash values. Most standard value-based classes implement a reasonable hash function: see `Float.hash`, `String.hash`, `List.hash`, ...

24.11.3 Prototypes

- [Comparable](#)
- [Container](#)
- [Object](#)
- [RangeIterable](#)

24.11.4 Construction

The Dictionary constructor takes arguments by pair (key, value).

urbiscript
Session

```
Dictionary.new("one", 1, "two", 2);
[00000000] ["one" => 1, "two" => 2]
Dictionary.new;
[00000000] [= > ]
```

There must be an even number of arguments.

urbiscript
Session

```
Dictionary.new("1", 2, "3");
[00000001:error] !!! new: odd number of arguments
```

You are encouraged to use the specific syntax for Dictionary literals:

urbiscript
Session

```
["one" => 1, "two" => 2];
[00000000] ["one" => 1, "two" => 2]
[=>];
[00000000] [= > ]
```

An extra comma can be added at the end of the list.

```
[  
    "one" => 1,  
    "two" => 2,  
];  
[00000000] ["one" => 1, "two" => 2]
```

urbiscript
Session

It is guaranteed that the pairs to insert are evaluated left-to-write, key first, the value.

```
["a".fresh => "b".fresh, "c".fresh => "d".fresh]  
== ["a_5" => "b_6", "c_7" => "d_8"];
```

Assertion
Block

24.11.5 Slots

- '==' (*that*)

Whether `this` equals `that`. This suppose that elements contained inside the dictionary are Comparable.

```
[ => ] == [ => ];  
["a" => 1, "b" => 2] == ["b" => 2, "a" => 1];
```

Assertion
Block

- '[]' (*key*)

Syntactic sugar for `get(key)`.

```
assert (["one" => 1]["one"] == 1);  
["one" => 1]["two"];  
[00000012:error] !!! missing key: two
```

urbiscript
Session

- '[]=' (*key, value*)

Syntactic sugar for `set(key, value)`, but returns `value`.

```
{  
    var d = ["one" =>"2"];  
    assert  
    {  
        (d["one"] = 1) == 1;  
        d["one"] == 1;  
    };  
};
```

urbiscript
Session

- `asBool`

Negation of `empty`.

```
[=>].asBool == false;  
["key" => "value"].asBool == true;
```

Assertion
Block

- `asList`

The contents of the dictionary as a `Pair` list (*key, value*).

```
["one" => 1, "two" => 2].asList == [(("one", 1), ("two", 2))];
```

Assertion
Block

Since Dictionary derives from `RangeIterable`, it is easy to iterate over a Dictionary using a range-for (Section 23.7.5.2). No particular order is ensured.

```
{  
    var res = [];  
    for| (var entry: ["one" => 1, "two" => 2])  
        res << entry.second;  
    assert(res == [1, 2]);  
};
```

urbiscript
Session

- **asString**

A string representing the dictionary. There is no guarantee on the order of the output.

Assertion Block

```
[=>].asString == "[ => ]";
["a" => 1, "b" => 2].asString == "[\"a\" => 1, \"b\" => 2]";
```

- **elementAdded**

An event emitted each time a new element is added to the Dictionary.

- **elementChanged**

An event emitted each time the value associated to a key of the Dictionary is changed.

- **elementRemoved**

An event emitted each time an element is removed from the Dictionary.

urbiscript Session

```
d = [ => ] |;
at(d.elementAdded?) echo ("added");
at(d.elementChanged?) echo ("changed");
at(d.elementRemoved?) echo ("removed");

d["key1"] = "value1";
[00000001] "value1"
[00000001] *** added

d["key2"] = "value2";
[00000001] "value2"
[00000001] *** added

d["key2"] = "value3";
[00000001] "value3"
[00000001] *** changed

d.erase("key2");
[00000002] ["key1" => "value1"]
[00000001] *** removed

d.clear;
[00000003] [ => ]
[00000001] *** removed

d.clear;
[00000003] [ => ]
```

- **clear**

Empty the dictionary.

Assertion Block

```
["one" => 1].clear.empty;
```

- **empty**

Whether the dictionary is empty.

Assertion Block

```
[=>].empty == true;
["key" => "value"].empty == false;
```

- **erase(*key*)**

Remove the mapping for *key*.

urbiscript Session

```
{
  var d = ["one" => 1, "two" => 2];
  assert
  {
    d.erase("two") === d;
```

```

    d == [ "one" => 1];
};

try
{
    [ "one" => 1, "two" => 2 ].erase("three");
    echo("never reached");
}
catch (var e if e.isA(Dictionary.KeyError))
{
    assert(e.key == "three")
};
}
;
```

- **get(key)**

The value associated to *key*. A `Dictionary.KeyError` exception is thrown if the key is missing.

```

var d = [ "one" => 1, "two" => 2 ] |;

assert(d.get("one") == 1);
[ "one" => 1, "two" => 2 ].get("three");
[00000010:error] !!! missing key: three

try
{
    d.get("three");
    echo("never reached");
}
catch (var e if e.isA(Dictionary.KeyError))
{
    assert(e.key == "three")
};
```

urbiscript
Session

- **getWithDefault(key, defaultValue)**

The value associated to *key* if it exists, *defaultValue* otherwise.

```

{
    var d = [ "one" => 1, "two" => 2 ];
    assert
    {
        d.getWithDefault("one", -1) == 1;
        d.getWithDefault("three", 3) == 3;
    };
}|;
```

urbiscript
Session

- **has(key)**

Whether the dictionary has a mapping for *key*.

```

{
    var d = [ "one" => 1 ];
    assert
    {
        d.has("one");
        !d.has("zero");
    };
}|;
```

urbiscript
Session

The infix operators `in` and `not in` use `has` (see Section 23.1.8.7).

```
"one" in      [ "one" => 1 ];
"two" not in [ "one" => 1 ];
```

Assertion
Block

- **init(*key1*, *value1*, ...)**

Insert the mapping from *key1* to *value1* and so forth.

urbiscript
Session

```
Dictionary.clone.init("one", 1, "two", 2);
[00000000] ["one" => 1, "two" => 2]
```

- **keys**

The list of all the keys. No particular order is ensured. Since [List](#) features the same function, uniform iteration over a List or a Dictionary is possible.

urbiscript
Session

```
{
  var d = ["one" => 1, "two" => 2];
  assert(d.keys == ["one", "two"]);
  assert({
    var res = [];
    for (var k: d.keys)
      res << d[k];
    res
  }
  == [1, 2]);
};
```

- **matchAgainst(*handler*, *pattern*)**

Pattern matching on members. See [Pattern](#).

urbiscript
Session

```
{
  // Match a subset of the dictionary.
  ["a" => var a] = ["a" => 1, "b" => 2];
  // get the matched value.
  assert(a == 1);
};
```

- **set(*key*, *value*)**

Map *key* to *value* and return [*this*](#) so that invocations to [set](#) can be chained. The possibly existing previous mapping is overridden.

urbiscript
Session

```
[=>].set("one", 2)
.set("two", 2)
.set("one", 1);
[00000000] ["one" => 1, "two" => 2]
```

- **size**

Number of element in the dictionary.

urbiscript
Session

```
{
  var d = [=>];
  assert(d.size == 0);
  d["a"] = 0;
  assert(d.size == 1);
  d["b"] = 1;
  assert(d.size == 2);
  d["a"] = 2;
  assert(d.size == 2);
};
```

24.12 Directory

A *Directory* represents a directory of the file system.

24.12.1 Prototypes

- `Object`

24.12.2 Construction

A `Directory` can be constructed with one argument: the path of the directory using a `String` or a `Path`. It can also be constructed by the method `open` of `Path`.

```
Directory.new(".");
[00000001] Directory(".");
Directory.new(Path.new("."));
[00000002] Directory(".")
```

urbiscript
Session

24.12.3 Slots

- `'/'(str)`

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

The `str String` is concatenated with the directory path. If the resulting path is either a directory or a file, `'/'` will returns either a `Directory` or a `File` object.

```
var dir1 = Directory.create("dir1")|;
var dir2 = Directory.create("dir1/dir2")|;
var file = File.create("dir1/file")|;
dir1 / "dir2";
[00000001] Directory("dir1/dir2")
dir1 / "file";
[00000002] File("dir1/file")
dir1.removeAll;
```

urbiscript
Session

- `'<<'(entity)`

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

If `entity` is a `Directory` or a `File`, `'<<'` copies `entity` in the `this` directory. Return `this` to allow chained operations.

```
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir2")|;
file = File.create("file")|;
dir1 << file << dir2;
[00000001] Directory("dir1")
dir1.content;
[00000003] ["dir2", "file"]
dir2;
[00000004] Directory("dir2")
file;
[00000005] File("file")
dir1.removeAll;
dir2.removeAll;
file.remove;
```

urbiscript
Session

- `asList`

The contents of the directory as a `Path` list. The various paths include the name of the directory `this`.

- **asString**

A **String** containing the path of the directory.

Assertion Block

```
Directory.new(".").asString == ".";
```

- **asPath**

A **Path** being the path of the directory.

- **content**

The contents of the directory as a **String** list. The strings include only the last component name; they do not contain the directory name of **this**.

- **copy(*dirname*)**

Copy recursively all items of the **this** directory into the directory *dirname* after creating it.

urbiscript Session

```
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir1/dir2")|;
file = File.create("dir1/file")|;
var directory1 = dir1.copy("directory1");
[00000001] Directory("directory1")
dir1;
[00000002] Directory("dir1")
directory1.content;
[00000003] ["dir2", "file"]
dir1.removeAll;
directory1.removeAll;
```

- **copyInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Copy **this** into *dirname* without creating it.

urbiscript Session

```
var dir = Directory.create("dir")|;
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir1/dir2")|;
file = File.create("dir1/file")|;
dir1.copyInto(dir);
[00000001] Directory("dir/dir1")
dir1;
[00000002] Directory("dir1")
dir1.content;
[00000003] ["dir2", "file"]
dir.content;
[00000004] ["dir1"]
Directory.new("dir/dir1").content;
[00000005] ["dir2", "file"]
dir.removeAll;
dir1.removeAll;
```

- **clear**

Remove all children recursively but not the directory itself. After a call to **clear**, a call to **empty** should return **true**.

urbiscript Session

```
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir1/dir2")|;
var file1 = File.create("dir1/file1")|;
var file2 = File.create("dir1/dir2/file2")|;
```

```

dir1.content;
[00000001] ["dir2", "file1"]
dir2.content;
[00000002] ["file2"]
dir1.clear;
assert(dir1.empty);
dir1.remove;

```

- **create(*name*)**

Create the directory *name* where *name* is either a [String](#) or a [Path](#). In addition to system errors that can occur, errors are raised if directory or file *name* already exists.

```

dir = Directory.new("dir");
[00000001:error] !!! new: directory does not exist: "dir"
dir = Directory.create("dir");
[00000002] Directory("dir")
dir = Directory.create("dir");
[00000001:error] !!! create: directory exists: "dir"
dir.content;
[00000003] []
dir.remove;

```

urbiscript
Session

- **createAll(*name*)**

Create the directory *name* where *name* is either a [String](#) or a [Path](#). If *name* is a path (or a [String](#) describing a path) no errors are raised if one directory doesn't exist or already exists. Instead [createAll](#) creates them all as in the Unix 'make -p' command.

```

Directory.create("dir1/dir2/dir3");
[00000001:error] !!! create: no such file or directory: "dir1/dir2/dir3"
dir1 = Directory.create("dir1");
[00000002] Directory("dir1")
Directory.createAll("dir1/dir2/dir3");
[00000002] Directory("dir1/dir2/dir3")
dir1.removeAll;

```

urbiscript
Session

- **empty**

Whether the directory is empty.

```

dir = Directory.create("dir")|;
assert(dir.empty);
File.create("dir/file")|;
assert(!dir.empty);
dir.removeAll;

```

urbiscript
Session

- **exists**

Whether the directory still exists.

```

dir = Directory.create("dir");
[00000001] Directory("dir")
assert(dir.exists);
dir.remove;
assert(!dir.exists);

```

urbiscript
Session

- **fileCreated(*name*)**

Event launched when a file is created inside the directory. May not exist if not supported by your architecture.

```

if (Path.new("./dummy.txt").exists)
    File.new("./dummy.txt").remove;

```

urbiscript
Session

```
{
    var d = Directory.new(".");
    waituntil(d.fileCreated?(var name));
    assert
    {
        name == "dummy.txt";
        Path.new(d.asString + "/" + name).exists;
    };
}
&
{
    sleep(100ms);
    File.create("./dummy.txt");
}|;
```

- **fileDeleted(*name*)**

Event launched when a file is deleted from the directory. May not exist if not supported by your architecture.

urbiscript
Session

```
if (!Path.new("./dummy.txt").exists)
    File.create("./dummy.txt")|;

{
    var d = Directory.new(".");
    waituntil(d.fileDeleted?(var name));
    assert
    {
        name == "dummy.txt";
        !Path.new(d.asString + "/" + name).exists;
    };
}
&
{
    sleep(100ms);
    File.new("./dummy.txt").remove;
}|;
```

- **basename**

Return a **String** containing the path of the directory without its dirname.

urbiscript
Session

```
var dir1 = Directory.create("dir1");
[00000001] Directory("dir1")
var dir2 = Directory.create("dir1/dir2");
[00000002] Directory("dir1/dir2")
dir1.basename;
[00000002] "dir1"
dir2.basename;
[00000003] "dir2"
dir1.removeAll;
```

- **lastModifiedDate**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Return a **Date** object stating when the directory was last modified.

- **moveInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Move `this` into `dirname` without creating it.

```
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir1/dir2")|;
var file = File.create("dir1/file")|;
var dir = Directory.create("dir")|;
dir1.moveTo(dir);
[00000001] Directory("dir/dir1")
dir1;
[00000002] Directory("dir/dir1")
dir1.content;
[00000003] ["dir2", "file"]
dir.content;
[00000004] ["dir1"]
dir.removeAll;
```

urbiscript
Session

- **parent**

Return the parent of the directory.

```
Directory.create("dir")|;
dir = Directory.create("dir/dir")|;
dir.parent;
[00000001] Directory("dir")
assert(dir.parent.parent.asString == Directory.current.asString);
dir.parent.removeAll;
```

urbiscript
Session

- **remove**

Remove the directory only if it is empty.

```
dir = Directory.create("dir")|;
File.create("dir/file")|;
dir.remove;
[00000001:error] !!! remove: directory not empty: "dir"
dir.clear;
dir.remove;
assert(!dir.exists);
```

urbiscript
Session

- **removeAll**

Remove all children recursively including the directory itself.

```
dir1 = Directory.create("dir1")|;
dir2 = Directory.create("dir1/dir2")|;
var file1 = File.create("dir1/file1")|;
var file2 = File.create("dir1/dir2/file2")|;
dir1.removeAll;
assert(!dir1.exists);
```

urbiscript
Session

- **rename**

Rename or move the directory.

```
dir = Directory.create("dir")|;
File.create("dir/file")|;
dir.rename("other");
[00000001] Directory("other")
dir;
[00000002] Directory("other")
dir.content;
[00000003] ["file"]
dir2 = Directory.create("dir2")|;
```

urbiscript
Session

```
dir.rename("dir2/other2");
[00000004] Directory("dir2/other2")
dir;
[00000005] Directory("dir2/other2")
dir.content;
[00000006] ["file"]
dir2.removeAll;
```

- **size**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

The size of all the directory content computed recursively.

urbiscript
Session

```
dir = Directory.create("dir")|;
Directory.create("dir/dir")|;
File.save("dir/file", "content");
file1 = File.create("dir/file")|;
File.save("dir/dir/file", "content");
file2 = File.create("dir/dir/file")|;
assert(dir.size() == file1.size() + file2.size());
```

24.13 Duration

This class records differences between [Dates](#).

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

24.13.1 Prototypes

- [Float](#)

24.13.2 Construction

Without argument, a null duration.

urbiscript
Session

```
Duration.new;
[00000001] Duration(0s)
Duration.new(1h);
[00023593] Duration(3600s)
```

Durations can be negative.

urbiscript
Session

```
Duration.new(-1);
[00000001] Duration(-1s)
```

24.13.3 Slots

- **asFloat**

Return the duration as a [Float](#).

Assertion
Block

```
Duration.new(1000).asFloat == 1000;
Duration.new(1000.1234).asFloat == 1000.1234;
```

- **asString**

Return the duration as a [String](#).

Assertion
Block

```
Duration.new(1000).asString == "1000s";
```

- **seconds**

Return the duration as a [Float](#).

```
Duration.new(1000) .seconds == 1000;
Duration.new(1000.52).seconds == 1000.52;
```

Assertion Block

24.14 Enumeration

Prototype of enumeration types.

24.14.1 Prototypes

- [RangeIterable](#)
- [Container](#)

24.14.2 Examples

See [Section 23.5](#).

24.14.3 Construction

An **Enumeration** is created with two arguments: the name of the enumeration type, and the list of possible values. Most of the time, it is a good idea to store it in a variable with the same name.

```
var Direction = Enumeration.new("Direction", ["up", "down", "left", "right"]);
[00000001] Direction
Direction.up;
[00000002] up
```

urbiscript Session

The following syntax is equivalent.

```
enum Direction
{
    up,
    down,
    left,
    right
};
[00000001] Direction
```

urbiscript Session

The created values are derive from the created enumeration type.

```
Direction.isA(Enumeration);
Direction.up.isA(Direction);
```

Assertion Block

24.14.4 Slots

- [asList](#)

Synonym for [values](#).

```
Direction.asList
== [Direction.up, Direction.down, Direction.left, Direction.right];
```

Assertion Block

Since it also derives from [RangeIterable](#), this enables all its features. For instance:

urbiscript Session

```
Direction.each(function (var d) { echo(d) });
[00000001] *** up
[00000001] *** down
[00000001] *** left
[00000001] *** right

for (var d in Direction)
    echo(d);
[00000001] *** up
[00000001] *** down
[00000001] *** left
[00000001] *** right

for| (var d in Direction)
    echo(d);
[00000001] *** up
[00000001] *** down
[00000001] *** left
[00000001] *** right

assert
{
    Direction.any(closure (var v) { v == Direction.up });
};
```

- **asString**

The name of the enumeration, or the name of the member if applied to a member.

Assertion Block

```
Direction.asString == "Direction";
Direction.up.asString == "up";
```

- **has(*v*)**

Whether *v* is a member of `this`. Since `Enumeration` derives from `Container`, it provides all its features.

urbiscript Session

```
enum CardinalDirection { north, east, south, west }|;

assert
{
    Direction.has(Direction.up);
    Direction.up in Direction;
    12 not in Direction;
    CardinalDirection.south not in Direction;
};
```

- **name**

The name of the enumeration. See also `asString`.

Assertion Block

```
Direction.name == "Direction";
Direction.up.name == "Direction";
```

- **size**

Number of possible values in the enumeration type. Equivalent to `values.size`.

Assertion Block

```
Direction.size == 4;
```

- **values**

The list of values.

Assertion Block

```
Direction.values
== [Direction.up, Direction.down, Direction.left, Direction.right];
```

24.15 Event

An *event* can be “emitted” and “caught”, or “sent” and “received”. See also [Section 15.2](#).

24.15.1 Prototypes

- [Object](#)

24.15.2 Examples

There are several examples of uses of events in the documentation of event-based constructs. See [at](#) ([Section 23.10.1](#)), [waituntil](#) ([Section 23.10.6](#)), [whenever](#) ([Section 23.10.7](#)), and so forth. The tutorial chapter about event-based programming contains other examples, see [Chapter 15](#).

24.15.3 Construction

An **Event** is created like any other object, without arguments.

```
var e = Event.new;
[00000001] Event_Ox9ad8118
```

urbiscript
Session

24.15.4 Slots

- **asEvent**
Return [this](#).

- **'emit'**

Throw an **Event**. This function is called by the bang operator. It takes any number of arguments, passed to the receiver when the event is caught. An event can also be emitted for a certain duration using `~`. The execution of [at](#) clauses etc., is asynchronous: the control flow might be released by the [emit](#) call before all the watchers have finished their execution.

- **onSubscribe**

This slot is not set by default. You can optionally assign an event to it. In this case, [onSubscribe](#) is triggered each time some code starts watching this event (by setting up an [at](#) or a [waituntil](#) on it for instance).

Throw a synchronized event. This call awaits that all functions that have to react to this event have returned. This function can have the same arguments as [emit](#).

- **trigger**

This function is used to launch an event during an unknown amount of time. Calling this function launches and keeps the event triggered and returns a handler object whose [stop](#) method stops launching the event. This method is asynchronous and the [stop](#) call will be asynchronous as well.

- **syncEmit**

This function does the same job as `'emit'` but the call will be synchronous.

- **syncTrigger**

This function does the same job as [trigger](#) but the call will be synchronous. The [stop](#) method of the handler object will be synchronous as well.

- **'||'(other)**

Logical or on events: a new Event that triggers whenever [this](#) or [other](#) triggers.

urbiscript
Session

```
var e1 = Event.new();
var e2 = Event.new();
var either = e1.'||'(e2);
at (either?)
  echo("!");
e1!;
[00000004] *** !
e2!;
[00000005] *** !
```

- `'<<' (other)`

Watch an *other* event status and reproduce it on itself, return `this`. This operator is similar to an optimized `||=` operator. Do not make events watch for themselves, directly or indirectly.

urbiscript
Session

```
var e3 = Event.new();
var e4 = Event.new();
var e_watch = Event.new << e3 << e4 |;
at (e_watch?)
  echo("!");
e3!;
[00000006] *** !
e4!;
[00000007] *** !
```

24.16 Exception

Exceptions are used to handle errors. More generally, they are a means to escape from the normal control-flow to handle exceptional situations.

The language support for throwing and catching exceptions (using `try/catch` and `throw`, see [Section 23.8](#)) work perfectly well with any kind of object, yet it is a good idea to throw only objects that derive from `Exception`.

24.16.1 Prototypes

- `Object`
- `Traceable`

24.16.2 Construction

There are several types of exceptions, each of which corresponding to a particular kind of error. The top-level object, `Exception`, takes a single argument: an error message.

urbiscript
Session

```
Exception.new("something bad has happened!");
[00000001] Exception 'something bad has happened!'
Exception.Arity.new("myRoutine", 1, 10, 23);
[00000002] Exception.Arity 'myRoutine: expected between 10 and 23 arguments, given 1'
```

24.16.3 Slots

Exception has many slots which are specific exceptions. See [Section 24.16.4](#) for their documentation.

- `backtrace`

The call stack at the moment the exception was thrown (not created), as a `List` of `StackFrames`, from the innermost to the outermost call. Uses `Traceable.backtrace`.

urbiscript
Session

```
//#push 1 "file.u"
try
{
    function innermost () { throw Exception.new("Ouch") };
    function inner      () { innermost() };
    function outer      () { inner() };
    function outermost () { outer() };

    outermost();
}
catch (var e)
{
    assert
    {
        e.backtrace[0].location.asString == "file.u:4.27-37";
        e.backtrace[0].name == "innermost";

        e.backtrace[1].location.asString == "file.u:5.27-33";
        e.backtrace[1].name == "inner";

        e.backtrace[2].location.asString == "file.u:6.27-33";
        e.backtrace[2].name == "outer";

        e.backtrace[3].location.asString == "file.u:8.3-13";
        e.backtrace[3].name == "outermost";
    };
};

//#pop
```

- **location**

The location from which the exception was thrown (not created).

```
eval("1/0");
[00090441:error] !!! 1.1-3: /: division by 0
[00090441:error] !!!      called from: eval
try
{
    eval("1/0");
}
catch (var e)
{
    assert (e.location.asString == "1.1-3");
}
```

urbiscript
Session

- **message**

The error message provided at construction.

```
Exception.new("Ouch").message == "Ouch";
```

Assertion
Block

24.16.4 Specific Exceptions

In the following, since these slots are actually Objects, what is presented as arguments to the slots are actually arguments to pass to the constructor of the corresponding exception type.

- **Argument(*routine*, *index*, *exception*)**

During the call of *routine*, the instantiation of the *index*-nth argument has thrown an *exception*.

```
Exception.Argument
    .new("myRoutine", 3, Exception.Type.new("19/11/2010", Date))
    .asString
== "myRoutine: argument 3: unexpected \"19/11/2010\", expected a Date";
```

Assertion
Block

- `ArgumentType(routine, index, effective, expected)`

Deprecated exception that derives from `Type`. The `routine` was called with a `index`-nth argument of type `effective` instead of `expected`.

Assertion Block

```
Exception.ArgumentType
  .new("myRoutine", 1, "hisResult", "Expectation")
  .asString
== "myRoutine: argument 1: unexpected \"hisResult\", expected a String";
[00000003:warning] !!! 'Exception.ArgumentType' is deprecated
```

- `Arity(routine, effective, min, max = void)`

The `routine` was called with an incorrect number of arguments (`effective`). It requires at least `min` arguments, and, if specified, at most `max`.

Assertion Block

```
Exception.Arity.new("myRoutine", 1, 10, 23).asString
== "myRoutine: expected between 10 and 23 arguments, given 1";
```

- `BadInteger(routine, fmt, effective)`

The `routine` was called with an inappropriate integer (`effective`). Use the format `fmt` to create an error message from `effective`. Derives from `BadNumber`.

Assertion Block

```
Exception.BadInteger.new("myRoutine", "bad integer: %s", 12).asString
== "myRoutine: bad integer: 12";
```

- `BadNumber(routine, fmt, effective)`

The `routine` was called with an inappropriate number (`effective`). Use the format `fmt` to create an error message from `effective`.

Assertion Block

```
Exception.BadNumber.new("myRoutine", "bad number: %s", 12.34).asString
== "myRoutine: bad number: 12.34";
```

- `Constness`

An attempt was made to change a constant value.

Assertion Block

```
Exception.Constness.new.asString
== "cannot modify const slot";
```

- `FileNotFoundException(name)`

The file named `name` cannot be found.

Assertion Block

```
Exception.FileNotFoundException.new("foo").asString
== "file not found: foo";
```

- `ImplicitTagComponent(msg)`

An attempt was made to create an implicit tag, a component of which being undefined.

Assertion Block

```
Exception.ImplicitTagComponent.new.asString
== "invalid component in implicit tag";
```

- `Lookup(object, name)`

A failed name lookup was performed on `object` to find a slot named `name`. Suggest what the user might have meant if `Exception.Lookup.fixSpelling` is true (which is the default).

Assertion Block

```
Exception.Lookup.new(Object, "GetSlot").asString
== "lookup failed: Object";
```

- **MatchFailure**

A pattern matching failed.

```
Exception.MatchFailure.new.asString
== "pattern did not match";
```

Assertion Block

- **NegativeNumber(*routine*, *effective*)**

The *routine* was called with a negative number (*effective*). Derives from [BadNumber](#).

```
Exception.NegativeNumber.new("myRoutine", -12).asString
== "myRoutine: unexpected -12, expected non-negative number";
```

Assertion Block

- **NonPositiveNumber(*routine*, *effective*)**

The *routine* was called with a non-positive number (*effective*). Derives from [BadNumber](#).

```
Exception.NonPositiveNumber.new("myRoutine", -12).asString
== "myRoutine: unexpected -12, expected positive number";
```

Assertion Block

- **Primitive(*routine*, *msg*)**

The built-in *routine* encountered an error described by *msg*.

```
Exception.Primitive.new("myRoutine", "cannot do that").asString
== "myRoutine: cannot do that";
```

Assertion Block

- **Redefinition(*name*)**

An attempt was made to refine a slot named *name*.

```
Exception.Redefinition.new("foo").asString
== "slot redefinition: foo";
```

Assertion Block

- **Scheduling(*msg*)**

Something really bad has happened with the Urbi task scheduler.

```
Exception.Scheduling.new("cannot schedule").asString
== "cannot schedule";
```

Assertion Block

- **Syntax(*loc*, *message*, *input*)**

Declare a syntax error in *input*, at location *loc*, described by *message*. *loc* is the location of the syntax error, *location* is the place the error was thrown. They are usually equal, except when the errors are caught while using [System.eval](#) or [System.load](#). In that case *loc* is really the position of the syntax error, while *location* refers to the location of the [System.eval](#) or [System.load](#) invocation.

```
Exception.Syntax
  .new(Location.new(Position.new("file.u", 14, 25)),
    "unexpected pouCharque", "file.u")
  .asString
== "file.u:14.25: syntax error: unexpected pouCharque";
```

Assertion Block

```
try
{
  eval("1 / / 0");
}
catch (var e)
{
  assert
  {
    e.isA(Exception.Syntax);
}
```

urbscript Session

```

    e.loc.asString == "1.5";
    e.input == "1 / / 0";
    e.message == "unexpected /";
}
];

```

- **Type(*effective*, *expected*)**

A value of type *effective* was received, while a value of type *expected* was expected.

Assertion Block

```

Exception.Type.new("hisResult", "Expectation").asString
== "unexpected \"hisResult\"", expected a String";

```

- **UnexpectedVoid**

An attempt was made to read the value of void.

Assertion Block

```

Exception.UnexpectedVoid.new.asString
== "unexpected void";

```

urbiscript Session

```

var a = void;
a;
[00000016:error] !!! unexpected void
[00000017:error] !!! lookup failed: a

```

24.17 Executable

This class is used only as a common ancestor to [Primitive](#) and [Code](#).

24.17.1 Prototypes

- [Object](#)

24.17.2 Construction

There is no point in constructing an Executable.

24.17.3 Slots

- [asExecutable](#)
Return [this](#).

24.18 File

24.18.1 Prototypes

- [Object](#)

24.18.2 Construction

Files may be created from a [String](#), or from a [Path](#). Using [new](#), the file must exist on the file system, and must be a file. You may use [create](#) to create a file that does not exist (or to override an existing one).

urbiscript Session

```

File.create("file.txt");
[00000001] File("file.txt")

File.new(Path.new("file.txt"));
[00000001] File("file.txt")

```

You may use [InputStream](#) and [OutputStream](#) to read or write to Files.

24.18.3 Slots

- **asList**

Read the file, and return its content as a list of its lines.

```
File.save("file.txt", "1\n2\n");
assert(File.new("file.txt").asList == ["1", "2"]);
```

urbiscript
Session

- **asPath**

A **Path** being the path of the file.

- **asPrintable**

```
File.save("file.txt", "1\n2\n");
assert(File.new("file.txt").asPrintable == "File(\"file.txt\")");
```

urbiscript
Session

- **asString**

The name of the opened file.

```
File.save("file.txt", "1\n2\n");
assert(File.new("file.txt").asString == "file.txt");
```

urbiscript
Session

- **basename**

Return a **String** containing the path of the file without its dirname.

- **content**

The content of the file as a **Binary** object.

```
File.save("file.txt", "1\n2\n");
assert
{
  File.new("file.txt").content == Binary.new("", "1\n2\n");
};
```

urbiscript
Session

- **copy(*filename*)**

Copy the file to a new file named *filename*.

```
File.save("file", "content");
var file = File.new("file");
[00000001] File("file")
var file2 = file.copy("file2");
[00000002] File("file2")
assert(file2.content == file.content);
file.remove;
file2.remove;
```

urbiscript
Session

- **copyInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Copy file into *dirname* directory.

```
var dir = Directory.create("dir")|;
file = File.create("file")|;
file.copyInto(dir);
[00000001] File("dir/file")
file;
[00000002] File("file")
```

urbiscript
Session

```
dir.content;
[00000003] ["file"]
dir.removeAll;
file.remove;
```

- **create(*name*)**

If the file *name* does not exist, create it and return a File to it. Otherwise, first empty it. See [OutputStream](#) for methods to add content to the file.

urbiscript
Session

```
var p = Path.new("create.txt") |
assert (!p.exists);

// Create the file, and put something in it.
var f = File.create(p)|;
var o = OutputStream.new(f)|;
o << "Hello, World!"|;
o.close;

assert
{
    // The file exists, with the expect contents.
    p.exists;
    f.content.data == "Hello, World!";

    // If we create is again, it is empty.
    File.create(p).isA(File);
    f.content.data == "";
};
```

- **lastModifiedDate**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Return a [Date](#) object stating when the file was last modified.

- **moveInto(*dirname*)**

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

Move file into *dirname* directory.

urbiscript
Session

```
dir = Directory.create("dir")|;
file = File.create("file")|;
file.moveInto(dir);
[00000001] File("dir/file")
file;
[00000001] File("dir/file")
dir.content;
[00000001] ["file"]
dir.removeAll;
```

- **remove**

Remove the current file. Returns void.

urbiscript
Session

```
var p = Path.new("foo.txt") |
p.exists;
[00000002] false
```

```
var f = File.create(p);
[00000003] File("foo.txt")
p.exists;
[00000004] true

f.remove;
p.exists;
[00000006] false
```

- **rename(*name*)**

Rename the file to *name*. If the target exists, it is replaced by the opened file. Return the file renamed.

```
File.save("file.txt", "1\n2\n");
File.new("file.txt").rename("bar.txt");
[00000001] File("bar.txt")
assert
{
    !Path.new("file.txt").exists;
    File.new("bar.txt").content.data == "1\n2\n";
};
```

urbiscript
Session

- **save(*name*, *content*)**

Use `create` to create the File named *name*, store the *content* in it, and close the file. Return void.

```
File.save("file.txt", "1\n2\n").isVoid;
File.new("file.txt").content.data == "1\n2\n";
```

Assertion
Block

- **size**

The size of the file.

```
File.save("file.txt", "1234");
File.new("file.txt").size;
[00000001] 4
```

urbiscript
Session

24.19 Finalizable

Objects that derive from this object will execute their `finalize` routine right before being destroyed (reclaimed) by the system. It is comparable to a *destructor*.

24.19.1 Example

The following object is set up to die verbosely.

```
var obj =
do (Finalizable.new)
{
    function finalize ()
    {
        echo ("Ouch");
    }
};
```

urbiscript
Session

It is reclaimed by the system when it is no longer referenced by any other object.

```
var alias = obj|;
obj = null;
```

urbiscript
Session

Here, the object is still alive, since `alias` references it. Once it no longer does, the object dies.

urbiscript
Session

```
alias = nil|;
[00000004] *** Ouch
```

24.19.2 Prototypes

- [Object](#)

24.19.3 Construction

The constructor takes no argument.

urbiscript
Session

```
Finalizable.new;
[00000527] Finalizable_0x135360
```

Because of specific constraints of `Finalizable`, you cannot change the prototype of an object to make it “finalizable”: it *must* be an instance of `Finalizable` from its inception.

There, instead of these two invalid constructs,

urbiscript
Session

```
class o1 : Finalizable.new
{
  function finalize()
  {
    echo("Ouch");
  }
}|;
[00000008:error] !!! apply: cannot inherit from a Finalizable without being one

class o2
{
  protos = [Finalizable];
  function finalize()
  {
    echo("Ouch");
  }
}|;
[00000010:error] !!! updateSlot: cannot inherit from a Finalizable without being one
```

write:

urbiscript
Session

```
var o3 =
  do (Finalizable.new)
  {
    function finalize()
    {
      echo("Ouch");
    }
  };
```

If you need multiple prototypes, do as follows.

urbiscript
Session

```
class Global.Foo
{
  function init()
  {
    echo("1");
  }
}};

class Global.FinalizableFoo
{
  addProto(Foo.new);
```

```

function 'new'()
{
  var r = clone |
  r.init |
  Finalizable.new.addProto(r);
};

function init()
{
  echo("2");
};

function finalize()
{
  echo("3");
};

};

var i = FinalizableFoo.new|;
[00000117] *** 1
[00000117] *** 2

i = nil;
[00000117] *** 3

```

24.19.4 Slots

- **finalize**

a simple function that takes no argument that will be evaluated when the object is re-claimed. Its return value is ignored.

```

Finalizable.new.setSlot("finalize", function() { echo("Ouch") })|;
[00033240] *** Ouch

```

urbiscript
Session

24.20 Float

A Float is a floating point number. It is also used, in the current version of urbiscript, to represent integers.

24.20.1 Prototypes

- [Comparable](#)
- [Orderable](#)
- [RangeIterable](#)

24.20.2 Construction

The most common way to create fresh floats is using the literal syntax. Numbers are composed of three parts:

integral (mandatory) a non empty sequence of (decimal) digits;

fractional (optional) a period, and a non empty sequence of (decimal) digits;

exponent (optional) either ‘e’ or ‘E’, an optional sign (‘+’ or ‘-’), then a non-empty sequence of digits.

In other words, float literals match the `[0-9]+(\.[0-9]+)?([eE][-+]?[0-9]+)?` regular expression. For instance:

```
Assertion Block
0 == 0000.0000;
// This is actually a call to the unary '+'.
+1 == 1;
0.123456 == 123456 / 1000000;
1e3 == 1000;
1e-3 == 0.001;
1.234e3 == 1234;
```

There are also some special numbers, `nan`, `inf` (see below).

```
Assertion Block
Math.log(0) == -inf;
Math.exp(-inf) == 0;
(inf/inf).isNaN;
```

A null float can also be obtained with `Float`'s `new` method.

```
Assertion Block
Float.new == 0;
```

24.20.3 Slots

- `abs`

Absolute value of the target.

```
Assertion Block
(-5).abs == 5;
0 .abs == 0;
5 .abs == 5;
```

- `acos`

Arccosine of the target.

```
Assertion Block
0.acos == Float.pi/2;
1.acos == 0;
```

- `asBool`

Whether non null.

```
Assertion Block
0.asBool == false;
0.1.asBool == true;
(-0.1).asBool == true;
inf.asBool == true;
nan.asBool == true;
```

- `asFloat`

Return the target.

```
Assertion Block
51.asFloat == 51;
```

- `asList`

Bounces to `seq`.

```
Assertion Block
3.asList == [0, 1, 2];
0.asList == [];
```

- `asin`

Arcsine of the target.

```
Assertion Block
0.asin == 0;
```

- **asString**

Return a string representing the target.

```
42.asString == "42";
42.51.asString == "42.51";
21474836470.asString == "21474836470";
4611686018427387904.asString == "4611686018427387904";
(-4611686018427387904).asString == "-4611686018427387904";
```

Assertion Block

- **atan**

Return the arctangent of the target.

```
0.atan == 0;
1.atan == Float.pi/4;
```

Assertion Block

- **'bitand'(that)**

The bitwise-and between `this` and `that`.

```
(3 bitand 6) == 2;
```

Assertion Block

- **'bitor'(that)**

Bitwise-or between `this` and `that`.

```
(3 bitor 6) == 7;
```

Assertion Block

- **ceil**

The smallest integral value greater than or equal to. See also `floor` and `round`.

```
0.ceil == 0;
1.4.ceil == 2;    1.5.ceil == 2;    1.6.ceil == 2;
(-1.4).ceil == -1; (-1.5).ceil == -1; (-1.6).ceil == -1;
inf.ceil == inf; (-inf).ceil == -inf;
nan.ceil.isNaN;
```

Assertion Block

- **clone**

Return a fresh Float with the same value as the target.

```
var x = 0;
[00000000] 0
var y = x.clone();
[00000000] 0
x === y;
[00000000] false
```

urbiscript Session

- **compl**

The complement to 1 of the target interpreted as a 32 bits integer.

```
compl 0 == 4294967295;
compl 4294967295 == 0;
```

Assertion Block

- **cos**

Cosine of the target.

```
0.cos == 1;
Float.pi.cos == -1;
```

Assertion Block

- **each(fun)**

Call the functional argument `fun` on every integer from 0 to target - 1, sequentially. The number must be non-negative.

Assertion Block

```
{
  var res = [];
  3.each(function (i) { res << 100 + i });
  res
}
== [100, 101, 102];

{
  var res = [];
  for(var x : 3) { res << x; sleep(20ms); res << (100 + x); };
  res
}
== [0, 100, 1, 101, 2, 102];

{
  var res = [];
  0.each (function (i) { res << 100 + i });
  res
}
== [];
```

- **'each|' (*fun*)** Call the functional argument *fun* on every integer from 0 to target - 1, with tight sequentiality. The number must be non-negative.

Assertion Block

```
{
  var res = [];
  3.'each|'(function (i) { res << 100 + i });
  res
}
== [100, 101, 102];

{
  var res = [];
  for|(var x : 3) { res << x; sleep(20ms); res << (100 + x); };
  res
}
== [0, 100, 1, 101, 2, 102];
```

- **'each&' (*fun*)**

Call the functional argument *fun* on every integer from 0 to target - 1, concurrently. The number must be non-negative.

Assertion Block

```
{
  var res = [];
  for& (var x : 3) { res << x; sleep(30ms); res << (100 + x) };
  res
}
== [0, 1, 2, 100, 101, 102];
```

- **exp**

Exponential of the target.

urbiscript Session

```
1.exp;
[00000000] 2.71828
```

- **floor**

the largest integral value less than or equal to `this`. See also `ceil` and `round`.

Assertion Block

```
0.floor == 0;
1.4.floor == 1;    1.5.floor == 1;    1.6.floor == 1;
(-1.4).floor == -2; (-1.5).floor == -2; (-1.6).floor == -2;
```

```
inf.floor == inf; (-inf).floor == -inf;
nan.floor.isNaN;
```

- **format(*finfo*)**

Format according to the [FormatInfo](#) object *finfo*. The precision, *finfo.precision*, sets the maximum number of digits after decimal point when in fixed or scientific mode, and in total when in default mode. Beware that 0 plays a special role, as it is not a “significant” digit.

Windows Issues

Under Windows the behavior differs slightly.

```
"%1.0d" % 0.1 == "0.1";
"%1.0d" % 1.1 == {if (System.Platform.isWindows) "1.1" else "1"};

"%1.0f" % 0.1 == "0";
"%1.0f" % 1.1 == "1";
```

Assertion Block

Conversion to hexadecimal requires [this](#) to be integral.

```
"%x" % 42 == "2a";
"%x" % 0xFFFF == "ffff";

"%x" % 0.5;
[00000005:error] !!! %: expected integer, got 0.5
```

Assertion Block

- **fresh**

Return a new integer at each call.

```
{
  var res = [];
  for (var i: 10)
    res << Float.fresh;
  assert (res == [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
  res = [];
  for (var i: 10)
    res << Float.fresh;
  assert (res == [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]);
}
```

urbscript Session

- **hash**

Return a [Hash](#) object corresponding to this float value. Two float hashes are equal if the float are equal. See [Object.hash](#).

```
0.hash.isA(Hash);
0.hash == 0.hash;
0.hash != 1.hash;
```

Assertion Block

- **hex**

A String with the conversion of [this](#) in hexadecimal. Requires [this](#) to be integral.

```
0.hex == "0";
0xFF.hex == "ff";
0xFFFF.hex == "ffff";
65535.hex == "ffff";
0xffffffff.hex == "ffffffff";

0.5.hex;
[00000005:error] !!! format: expected integer, got 0.5
```

Assertion Block

- **inf**

Return the infinity.

urbiscript
Session

```
Float.inf;
[00000000] inf
```

- **isInf**

Whether is infinite.

Assertion
Block

```
!0.isInf; !1.isInf; !(-1).isInf;
!nan.isInf;
inf.isInf; (-inf).isInf;
```

- **isNaN**

Whether is NaN.

Assertion
Block

```
!0.isnan; !1.isnan; !(-1).isnan;
!inf.isnan; !(-inf).isnan;
nan.isnan;
```

- **limits**

See [Float.limits](#).

Assertion
Block

- **log**

The logarithm of the target.

```
0.log == -inf;
1.log == 0;
1.exp.log == 1;
```

- **max(arg1, ...)**

Bounces to [List.max](#) on [this, arg1, ...].

Assertion
Block

```
1.max == 1;
1.max(2, 3) == 3;
3.max(1, 2) == 3;
```

- **min(arg1, ...)**

Bounces to [List.min](#) on [this, arg1, ...].

Assertion
Block

```
1.min == 1;
1.min(2, 3) == 1;
3.min(1, 2) == 1;
```

- **nan**

The “not a number” special float value. More precisely, this returns the “quiet NaN”, i.e., it is propagated in the various computations, it does not raise exceptions.

urbiscript
Session

```
Float.nan;
[00000000] nan
(Float.nan + Float.nan) / (Float.nan - Float.nan);
[00000000] nan
```

A NaN has one distinctive property over the other Floats: it is equal to no other float, not even itself. This behavior is mandated by the [IEEE 754-2008](#) standard.

Assertion
Block

```
{ var n = Float.nan; n === n};
{ var n = Float.nan; n != n};
```

- **pi**

π .

```
Float.pi.cos ** 2 + Float.pi.sin ** 2 == 1;
```

Assertion Block

- **random**

A random integer between 0 (included) and the target (excluded).

```
20.map(function (dummy) { 5.random });
[00000000] [1, 2, 1, 3, 2, 3, 2, 2, 4, 4, 4, 1, 0, 0, 0, 3, 2, 4, 3, 2]
```

urbiscript Session

- **round**

The integral value nearest to `this` rounding half-way cases away from zero. See also `ceil` and `floor`.

```
0.round == 0;
1.4.round == 1;    1.5.round == 2;    1.6.round == 2;
(-1.4).round == -1; (-1.5).round == -2; (-1.6).round == -2;
inf.round == inf; (-inf).round == -inf;
nan.round.isNaN;
```

Assertion Block

- **seq**

The sequence of integers from 0 to `this`-1 as a list. The number must be non-negative.

```
3.seq == [0, 1, 2];
0.seq == [];
```

Assertion Block

- **sign**

Return 1 if `this` is positive, 0 if it is null, -1 otherwise.

```
(-1164).sign == -1;
0.sign == 0;
(1164).sign == 1;
```

Assertion Block

- **sin**

The sine of the target.

```
0.sin == 0;
```

Assertion Block

- **sqr**

Square of the target.

```
32.sqr == 1024;
32.sqr == 32 ** 2;
```

Assertion Block

- **sqrt**

The square root of the target.

```
1024.sqrt == 32;
1024.sqrt == 1024 ** 0.5;
```

Assertion Block

- **srandom**

Initialized the seed used by the random function. As opposed to common usage, you should not use

urbiscript Session

```
{
    var now = Date.now.timestamp;
    now.random;
    var list1 = 20.map(function (dummy) { 5.random });
    now.random;
    var list2 = 20.map(function (dummy) { 5.random });
    assert
    {
        list1 == list2;
    }
};
```

- **tan**

Tangent of the target.

urbiscript
Session

```
assert(0.tan == 0);
(Float.pi/4).tan;
[00000000] 1
```

- **times(*fun*)**

Call the functional argument *fun* **this** times.

urbiscript
Session

```
3.times(function () { echo("ping") });
[00000000] *** ping
[00000000] *** ping
[00000000] *** ping
```

- **trunc**

Return the target truncated.

Assertion
Block

```
1.9.trunc == 1;
(-1.9).trunc == -1;
```

- **'^'(*that*)**

Bitwise exclusive or between **this** and *that*.

Assertion
Block

```
(3 ^ 6) == 5;
```

- **'>>'(*that*)**

this shifted by *that* bits towards the right.

Assertion
Block

```
4 >> 2 == 1;
```

- **'<'(*that*)**

Whether **this** is less than *that*. The other comparison operators (**<=**, **>**, ...) can thus also be applied on floats since *Float* inherits **Orderable**.

Assertion
Block

```
0 < 1;
!(1 < 0);
!(0 < 0);
```

- **'<<'(*that*)**

this shifted by *that* bit towards the left.

Assertion
Block

```
4 << 2 == 16;
```

- **'-'(*that*)**

this subtracted by *that*.

Assertion
Block

```
6 - 3 == 3;
```

- `'+'(that)`

The sum of `this` and `that`.

```
1 + 1 == 2;
```

Assertion
Block

- `'/'(that)`

The quotient of `this` divided by `that`.

```
50 / 10 == 5;
10 / 50 == 0.2;
```

Assertion
Block

- `'%'(that)`

`this` modulo `that`.

```
50 % 11 == 6;
```

Assertion
Block

- `'*(that)`

Product of `this` by `that`.

```
2 * 3 == 6;
```

Assertion
Block

- `'**'(that)`

`this` to the `that` power ($this^{that}$).

```
2 ** 10 == 1024;
2 ** 31 == 2147483648;
-2 ** 31 == -2147483648; // This is -(2**31).
2 ** 32 == 4294967296;
-2 ** 32 == -4294967296; // This is -(2**32).
```

Assertion
Block

- `'=='(that)`

Whether `this` equals `that`.

```
1 == 1;
!(1 == 2);
```

Assertion
Block

24.21 Float.limits

This singleton handles various limits related to the `Float` objects.

24.21.1 Slots

- `digits`

Number of digits (in `radix` base) in the mantissa.

```
Float.limits.digits;
```

Assertion
Block

- `digits10`

Number of digits (in decimal base) that can be represented without change.

```
Float.limits.digits10;
```

Assertion
Block

- **epsilon**

Machine epsilon (the difference between 1 and the least value greater than 1 that is representable).

Assertion Block

```
1 != 1 + Float.limits.epsilon;
1 == 1 + Float.limits.epsilon / 2;
```

- **max**

Maximum finite value.

Assertion Block

```
Float.limits.max      != Float.inf;
Float.limits.max * 2 == Float.inf;
```

- **maxExponent**

Maximum integer value for the exponent that generates a normalized floating-point number.

Assertion Block

```
Float.inf != Float.limits.radix ** (Float.limits.maxExponent - 1);
Float.inf == Float.limits.radix ** Float.limits.maxExponent;
```

- **maxExponent10**

Maximum integer value such that 10 raised to that power generates a normalized finite floating-point number.

Assertion Block

```
Float.inf != 10 ** Float.limits.maxExponent10;
Float.inf == 10 ** (Float.limits.maxExponent10 + 1);
```

- **min**

Minimum positive normalized value.

Assertion Block

```
0 != Float.limits.min;
```

- **minExponent**

Minimum negative integer value for the exponent that generates a normalized floating-point number.

Assertion Block

```
0 != Float.limits.radix ** Float.limits.minExponent;
```

- **minExponent10**

Minimum negative integer value such that 10 raised to that power generates a normalized floating-point number.

Assertion Block

```
0 != 10 ** Float.limits.minExponent10;
```

- **radix**

Base of the exponent of the representation.

Assertion Block

```
Float.limits.radix == 2;
```

24.22 FormatInfo

A *format info* is used when formatting a la `printf`. It store the formatting pattern itself and all the format information it can extract from the pattern.

24.22.1 Prototypes

- **Object**

24.22.2 Construction

The constructor expects a string as argument, whose syntax is similar to `printf`'s. It is detailed below.

```
var f = FormatInfo.new("%+2.3d");
[00000001] %+2.3d
```

urbscript
Session

A formatting pattern must one of the following (brackets denote optional arguments):

- `%options spec`
- `%|options[spec]|`

options is a sequence of 0 or several of the following characters:

‘-’	Left alignment.
‘=’	Centered alignment.
‘+’	Show sign even for positive number.
‘ ’	If the string does not begin with ‘+’ or ‘-’, insert a space before the converted string.
‘0’	Pad with 0's (inserted after sign or base indicator).
‘#’	Show numerical base, and decimal point.

spec is the conversion character and must be one of the following:

‘s’	Default character, prints normally
‘d’	Case modifier: lowercase
‘D’	Case modifier: uppercase
‘x’	Prints in hexadecimal lowercase
‘X’	Prints in hexadecimal uppercase
‘o’	Prints in octal
‘e’	Prints floats in scientific format
‘E’	Prints floats in scientific format uppercase
‘f’	Prints floats in fixed format

24.22.3 Slots

- `alignment`

Requested alignment: -1 for left, 0 for centered, 1 for right (default).

```
FormatInfo.new("%s").alignment == 1;
FormatInfo.new("%=s").alignment == 0;
FormatInfo.new("%-s").alignment == -1;
```

Assertion
Block

- `alt`

Whether the “alternative” display is requested (‘#’).

```
FormatInfo.new("%s").alt == false;
FormatInfo.new("%#s").alt == true;
```

Assertion
Block

- `group`

Separator to use for thousands. Corresponds to the ‘,’ *option*.

```
FormatInfo.new("%s").group == "";
FormatInfo.new("%'s").group == " ";
```

Assertion
Block

- `pad`

The padding character to use for alignment requests. Defaults to space.

Assertion
Block

```
FormatInfo.new("%s").pad == " ";
FormatInfo.new("%0s").pad == "0";
```

- **pattern**

The pattern given to the constructor.

Assertion Block

```
FormatInfo.new("%#'12.8s").pattern == "%#'12.8s";
```

- **precision**

When formatting a [Float](#), the maximum number of digits after decimal point when in fixed or scientific mode, and in total when in default mode. When formatting other objects with spec-char ‘s’, the conversion string is truncated to the precision first chars. The eventual padding to `width` is done after truncation.

Assertion Block

```
FormatInfo.new("%s").precision == 6;
FormatInfo.new("%23.3s").precision == 3;
```

- **prefix**

The string to display before positive numbers. Defaults to empty.

Assertion Block

```
FormatInfo.new("%s").prefix == "";
FormatInfo.new("% s").prefix == " ";
FormatInfo.new("%+s").prefix == "+";
```

- **spec**

The specification character, regardless of the case conversion requests.

Assertion Block

```
FormatInfo.new("%s").spec == "s";
FormatInfo.new("%23.3s").spec == "s";
FormatInfo.new("%'X").spec == "x";
```

- **uppercase**

Case conversion: -1 for lower case, 0 for no conversion (default), 1 for conversion to uppercase. The value depends on the case of specification character, except for ‘%s’ which corresponds to 0.

Assertion Block

```
FormatInfo.new("%s").uppercase == 0;
FormatInfo.new("%d").uppercase == -1;
FormatInfo.new("%D").uppercase == 1;
FormatInfo.new("%x").uppercase == -1;
FormatInfo.new("%X").uppercase == 1;
```

- **width**

Width requested for alignment.

Assertion Block

```
FormatInfo.new("%s").width == 0;
FormatInfo.new("%10s").width == 10;
```

24.23 Formatter

A *formatter* stores format information of a format string like used in `printf` in the C library or in `boost::format`.

24.23.1 Prototypes

- [Object](#)

24.23.2 Construction

Formatters are created with the format string. It cuts the string to separate regular parts of string and formatting patterns, and stores them.

```
Formatter.new("Name:%s, Surname:%s;");
[00000001] Formatter ["Name:", %s, "Surname:", %s, ";"]
```

urbscript
Session

Actually, formatting patterns are translated into `FormatInfo`.

24.23.3 Slots

- `asList`

Return the content of the `formatter` as a list of strings and `FormatInfo`.

```
Formatter.new("Name:%s, Surname:%s;").asListasString
== "[\"Name:\", %s, \", Surname:\", %s, \";\"]";
```

Assertion
Block

- `'%'(args)`

Use `this` as format string and `args` as the list of arguments, and return the result (a `String`). The arity of the Formatter (i.e., the number of expected arguments) and the size of `args` must match exactly.

This operator concatenates regular strings and the strings that are result of `asString` called on members of `args` with the appropriate `FormatInfo`.

```
Formatter.new("Name:%s, Surname:%s;") % ["Foo", "Bar"]
== "Name:Foo, Surname:Bar;";
```

Assertion
Block

If `args` is not a `List`, then the call is equivalent to calling `'%'([args])`.

```
Formatter.new("%06.3f") % Math.pi
== "03.142";
```

Assertion
Block

Note that `String.%'` provides a nicer interface to this operator:

```
%06.3f % Math.pi == "03.142";
```

Assertion
Block

It is nevertheless interesting to use the Formatter for performance reasons if the format is reused many times.

```
{
    // Some large database of people.
    var people =
        [["Foo", "Bar"], [
            ["One", "Two"],
            ["Un", "Deux"]]];
    var f = Formatter.new("Name:%7s, Surname:%7s;");
    for (var p: people)
        echo (f % p);
};

[00031939] *** Name:     Foo, Surname:     Bar;
[00031940] *** Name:     One, Surname:     Two;
[00031941] *** Name:     Un, Surname:     Deux;
```

urbscript
Session

24.24 Global

`Global` is designed for the purpose of being global namespace. Since `Global` is a prototype of `Object` and all objects are an `Object`, all slots of `Global` are accessible from anywhere.

24.24.1 Prototypes

- [uobjects](#).
- [Tag.tags](#) (see [Tag](#))
- [Math](#)
- [System](#)
- [Object](#)

24.24.2 Slots

- [Barrier](#)
See [Barrier](#).
- [Binary](#)
See [Binary](#).
- [CallMessage](#)
See [CallMessage](#).
- [cerr](#)
A predefined stream for error messages. Strings sent to it are not escaped, contrary to regular streams (see [output](#) for instance).

urbiscript
Session

```
cerr << "Message";
[00015895:error] Message
cerr << "\"quote\"";
[00015895:error] "quote"
```

- [Channel](#)
See [Channel](#).
- [clog](#)
A predefined stream for log messages. Strings are output escaped.

urbiscript
Session

```
clog << "Message";
[00015895:clog] "Message"
```

- [Code](#)
See [Code](#).
- [Comparable](#)
See [Comparable](#).
- [cout](#)
A predefined stream for output messages. Strings are output escaped.

urbiscript
Session

```
cout << "Message";
[00015895:output] "Message"
cout << "\"quote\"";
[00015895:output] "\"quote\""
```

- [Date](#)
See [Date](#).
- [detach\(*exp*\)](#)
Bounce to [Control.detach](#), see [Control](#).

- **Dictionary**

See [Dictionary](#).

- **Directory**

See [Directory](#).

- **disown(*exp*)**

Bounce to [Control.disown](#), see [Control](#).

- **Duration**

See [Duration](#).

- **echo(*value*, *channel* = "")**

Bounce to [lobby.echo](#), see [Lobby.echo](#).

```
echo("111", "foo");
[00015895:foo] *** 111
echo(222, "");
[00051909] *** 222
echo(333);
[00055205] *** 333
```

urbiscript
Session

- **evaluate**

This [UVar](#) provides a synchronous interface to the Urbi engine: write to it to “send” an expression to compute it, and “read” it to get the result. This UVar is designed to be used from the C++; it makes little sense in urbiscript, use [System.eval](#) instead, if it is really required (see [Section 20.1](#)). Since the semantics of the assignment requires that it evaluates to the right-hand side argument, reading **evaluate** after the assignment is needed, which makes race conditions likely. To avoid this, use `|` (or better yet, do not use **evaluate** at all in urbiscript).

```
(evaluate = "1+2;") == "1+2;";
evaluate == 3;
{ evaluate = "1+2;" | evaluate } == 3;
{ evaluate = "var x = 1;"; x } == 1;
```

Assertion
Block

Errors raise an exception.

```
evaluate = "1/0;";
[00087671:error] !!! Exception caught while processing notify on Global.evaluate:
[00087671:error] !!! 1.1-3: /: division by 0
[00087671:error] !!!      called from: updateSlot
[00087671] "1/0;"
```

urbiscript
Session

- **Event**

See [Event](#).

- **Exception**

See [Exception](#).

- **Executable**

See [Executable](#).

- **external**

An system object used to implement UObject support in urbiscript.

- **false**

See [Section 24.3.3](#).

- **File**

See [File](#).

- **Finalizable**

See [Finalizable](#).

- **Float**

See [Float](#).

- **FormatInfo**

See [FormatInfo](#).

- **Formatter**

See [Formatter](#).

- **getProperty(*slotName*, *propName*)**

This wrapper around [Object.getProperty](#) is actually a by-product of the existence of the [evaluate UVar](#).

- **Global**

See [Global](#).

- **Group**

See [Group](#).

- **InputStream**

See [InputStream](#).

- **isdef(*qualifiedIdentifier*)**

Whether the *qualifiedIdentifier* is defined. It features some (fragile) magic to support an argument passed as a literal (`isdef(foo)`), not a string (`isdef("foo")`). It is not recommended to use this feature, which is provided for urbiscipt compatibility. See [Object.hasLocalSlot](#) and [Object.hasSlot](#) for safer alternatives.

urbiscipt
Session

```
assert
{
    !isdef(a);
    !isdef(a.b);
    !isdef(a.b.c);
};

var a = Object.new();
assert
{
    isdef(a);
    !isdef(a.b);
    !isdef(a.b.c);
};

var a.b = Object.new();
assert
{
    isdef(a);
    isdef(a.b);
    !isdef(a.b.c);
};

var a.b.c = Object.new();
assert
{
    isdef(a);
    isdef(a.b);
```

```
    isdef(a.b.c);
};
```

- **Job**
See [Job](#).
- **Kernel1**
See [Kernel1](#).
- **Lazy**
See [Lazy](#).
- **List**
See [List](#).
- **Loadable**
See [Loadable](#).
- **Lobby**
See [Lobby](#).
- **Math**
See [Math](#).

● **methodToFunction(*name*)**

Create a function from the method *name* so that calling the function which arguments (*a*, *b*, ...) is that same as calling *a.name(b, ...)*.

```
var uid_of = methodToFunction("uid")|;
assert
{
  uid_of(Object) == Object.uid;
  uid_of(Global) == Global.uid;
};
var '+_of' = methodToFunction("+")|;
assert
{
  '+_of'( 1, 2) == 1 + 2;
  '+_of'("1", "2") == "1" + "2";
  '+_of'([1], [2]) == [1] + [2];
};
```

urbiscript
Session

- **Mutex**
See [Mutex](#).
- **nil**
See [nil](#).
- **Object**
See [Object](#).
- **Orderable**
See [Orderable](#).
- **OutputStream**
See [OutputStream](#).
- **Pair**
See [Pair](#).

- **Path**
See [Path](#).
- **Pattern**
See [Pattern](#).
- **persist(*exp*)**
Bounce to [Control.persist](#), see [Control](#).
- **Position**
See [Position](#).
- **Primitive**
See [Primitive](#).
- **Process**
See [Process](#).
- **Profile**
See [Profile](#).
- **PseudoLazy**
See [PseudoLazy](#).
- **PubSub**
See [PubSub](#).
- **RangeIterable**
See [RangeIterable](#).
- **Regexp**
See [Regexp](#).
- **Semaphore**
See [Semaphore](#).
- **Server**
See [Server](#).
- **Singleton**
See [Singleton](#).
- **Socket**
See [Socket](#).
- **String**
See [String](#).
- **System**
See [System](#).
- **Tag**
See [Tag](#).
- **Timeout**
See [Timeout](#).
- **TrajectoryGenerator**
See [TrajectoryGenerator](#).

- **Triplet**
See [Triplet](#).
- **true**
See [Section 24.3.3](#).
- **Tuple**
See [Tuple](#).
- **UObject**
See [UObject](#).
- **uobjects**
An object whose slots are all the [UObject](#) bound into the system. See [uobjects](#).
- **UValue**
See [UValue](#).
- **UVar**
See [UVar](#).
- **void**
See [void](#).
- **wall(*value*, *channel* = "")**
Bounce to [lobby.wall](#), see [Lobby](#).

```
wall("111", "foo");
[00015895:foo] *** 111
wall(222, "");
[00051909] *** 222
wall(333);
[00055205] *** 333
```

urbiscript
Session

- **warn(*message*)**
Issue *message* on [Channel.warning](#).

```
warn("cave canem");
[00015895:warning] !!! cave canem
```

urbiscript
Session

24.25 Group

A transparent means to send messages to several objects as if they were one.

24.25.1 Example

The following session demonstrates the features of the Group objects. It first creates the [Sample](#) family of object, makes a group of such object, and uses that group.

```
class Sample
{
    var value = 0;
    function init(v) { value = v; };
    function asString() { "<" + value.asString + ">"; };
    function timesTen() { new(value * 10); };
    function plusTwo() { new(value + 2); };
};

[00000000] <0>

var group = Group.new(Sample.new(1), Sample.new(2));
```

urbiscript
Session

```
[00000000] Group [<1>, <2>]
group << Sample.new(3);
[00000000] Group [<1>, <2>, <3>]
group.timesTen.plusTwo;
[00000000] Group [<12>, <22>, <32>]

// Bouncing getSlot and updateSlot.
group.value;
[00000000] Group [1, 2, 3]
group.value = 10;
[00000000] Group [10, 10, 10]

// Bouncing to each&.
var sum = 0|
for& (var v : group)
  sum += v.value;
sum;
[00000000] 30
```

24.25.2 Prototypes

- `Object`

24.25.3 Construction

Groups are created like any other object. The constructor can take members to add to the group.

urbiscript
Session

```
Group.new;
[00000000] Group []
Group.new(1, "two");
[00000000] Group [1, "two"]
```

24.25.4 Slots

- `add(member, ...)`
Add members to `this` group, and return `this`.
- `asString`
Report the members.
- `each(action)`
Apply `action` to all the members, in sequence, then return the Group of the results, in the same order. Allows to iterate over a Group via `for`.
- `each&(action)`
Apply `action` to all the members, concurrently, then return the Group of the results. The order is *not* necessarily the same. Allows to iterate over a Group via `for&`.
- `fallback`
This function is called when a method call on `this` failed. It bounces the call to the members of the group, collects the results returned as a group. This allows to chain grouped operation in a row. If the dispatched calls return `void`, returns a single `void`, not a “group of `void`”.
- `getProperty(slot, prop)`
Bounced to the members so that `this.slot->prop` actually collects the values of the property `prop` of the slots `slot` of the group members.

- **hasProperty(*name*)**
Bounced to the members.
- **hasSlot(*name*)**
True if and only if all the members have the slot.

```
var g = Group.new(1, 2);
[00000000] Group [1, 2]
g.hasSlot("foo");
[00000000] false
g.hasSlot("+");
[00000000] true
g + 1;
[00000000] Group [2, 3]
```

urbiscript
Session

- **remove(*member*, ...)**
Remove members from **this** group, and return **this**.
- **setProperty(*slot*, *prop*, *value*)**
Bounced to the members so that **this.slot->prop = value** actually updates the value of the property *prop* in the slots *slot* of the group members.
- **updateSlot(*name*, *value*)**
Bounced to the members so that **this.name = value** actually updates the value of the slot *name* in the group members.
- **'<<'(*member*)**
Syntactic sugar for **add**.

24.26 Hash

A *hash* is a condensed, easily comparable representation of another value. They are mainly used to map [Dictionary](#) keys to values.

Equal objects must always have the same hash. Different objects should, as much as possible, have different hashes.

24.26.1 Prototypes

- [Object](#)

24.26.2 Construction

Objects can be hashed with [Object.hash](#).

```
Object.new.hash.isA(Hash);
```

Assertion
Block

24.26.3 Slots

- **asFloat**
A Float value equivalent to the Hash object. Two hashes have the same Float representation if and only if they are equal.

```
var h1 = Object.new.hash();
var h2 = Object.new.hash();
assert
{
  h1.asFloat == h1.asFloat;
  h1.asFloat != h2.asFloat;
};
```

urbiscript
Session

- `combine(that)`

Combine `that`'s hash with `this`, and return `this`. This is used to hash composite objects based on more primitive object hashes. For instance, an object with two slots could be hashed by hashing its first first, and combining the second in.

urbiscript
Session

```
class C
{
    function init(var a, var b)
    {
        var this.a = a;
        var this.b = b;
    };

    function hash()
    {
        this.a.hash().combine(b)
    };
}

assert
{
    C.new(0, 0).hash == C.new(0, 0).hash;
    C.new(0, 0).hash != C.new(0, 1).hash;
};
```

24.27 InputStream

InputStreams are used to read (possibly binary) files by hand. `File` provides means to swallow a whole file either as a single large string, or a list of lines. `InputStream` provides a more fine-grained interface to read files.

24.27.1 Prototypes

- `Stream`

Windows Issues

Beware that because of limitations in the current implementation, one cannot safely read from two different files at the same time under Windows.

24.27.2 Construction

An InputStream is a reading-interface to a file, so its constructor requires a `File`.

urbiscript
Session

```
File.save("file.txt", "1\n2\n");
var is = InputStream.new(File.new("file.txt"));
[00000001] InputStream_0x827000
```

Bear in mind that open streams should be closed ([Section 24.61.2](#)).

urbiscript
Session

```
is.close;
```

24.27.3 Slots

- `get`

Get the next available byte as a `Float`, or `void` if the end of file was reached. Raise an error if the file is closed.

urbiscript
Session

```
{
    File.save("file.txt", "1\n2\n");
    var i = InputStream.new(File.new("file.txt"));
    var x;
    while (!(x = i.get.acceptVoid).isVoid)
        cout << x;
    i.close;
    i.get;
};
[00000001:output] 49
[00000002:output] 10
[00000003:output] 50
[00000004:output] 10
[00000005:error] !!! get: stream is closed
```

- **getChar**

Get the next available byte as a **String**, or **void** if the end of file was reached. Raise an error if the file is closed.

```
{
    File.save("file.txt", "1\n2\n");
    var i = InputStream.new(File.new("file.txt"));
    var x;
    while (!(x = i.getChar.acceptVoid).isVoid)
        cout << x;
    i.close;
    i.getChar;
};
[00000001:output] "1"
[00000002:output] "\n"
[00000003:output] "2"
[00000004:output] "\n"
[00000005:error] !!! getChar: stream is closed
```

urbiscript
Session

- **getLine**

Get the next available line as a **String**, or **void** if the end of file was reached. The end-of-line characters are trimmed. Raise an error if the file is closed.

```
{
    File.save("file.txt", "1\n2\n");
    var i = InputStream.new(File.new("file.txt"));
    var x;
    while (!(x = i.getLine.acceptVoid).isVoid)
        cout << x;
    i.close;
    i.getLine;
};
[00000001:output] "1"
[00000002:output] "2"
[00000005:error] !!! getLine: stream is closed
```

urbiscript
Session

24.28 IoService

A **IoService** is used to manage the various operations of a set of **Socket**.

All **Socket** and **Server** are by default using the default **IoService** which is polled regularly by the system.

24.28.1 Example

Using a different **IoService** is required if you need to perform synchronous read operations.

The `Socket` must be created by the `IoService` that will handle it using its `makeSocket` function.

urbiscript
Session

```
var io = IoService.new();
var s = io.makeSocket();
```

You can then use this socket like any other.

urbiscript
Session

```
// Make a simple hello server.
var serverPort = 0
do(Server.new)
{
    listen("127.0.0.1", "0");
    lobby.serverPort = port;
    at(connection?(var s))
    {
        s.write("hello");
    }
};

// Connect to it using our socket.
s.connect("0.0.0.0", serverPort);
at(s.received?(var data))
    echo("received something");
s.write("1");
```

... except that nothing will be read from the socket unless you call one of the `poll` functions of `io`.

urbiscript
Session

```
sleep(200ms);
s.isConnected() // Nothing was received yet
[00000001] true
io.poll();
[00000002] *** received something
sleep(200ms);
```

24.28.2 Prototypes

- `Object`

24.28.3 Construction

A `IoService` is constructed with no argument.

24.28.4 Slots

- `makeServer`
Create and return a new `Server` using this `IoService`.
- `makeSocket`
Create and return a new `Socket` using this `IoService`.
- `poll`
Handle all pending socket operations(read, write, accept) that can be performed without waiting.
- `pollFor(duration)`
Will block for `duration` seconds, and handle all ready socket operations during this period.
- `pollOneFor(duration)`
Will block for at most `duration`, and handle the first ready socket operation and immediately return.

24.29 Job

Jobs are independent threads of executions. Jobs can run concurrently. They can also be managed using [Tags](#).

24.29.1 Prototypes

- [Object](#)
- [Traceable](#)

24.29.2 Construction

A Job is typically constructed via [Control.detach](#), [Control.disown](#), or [System.spawn](#).

```
detach(sleep(10));
[00202654] Job<shell_4>

disown(sleep(10));
[00204195] Job<shell_5>

spawn (function () { sleep(10) }, false);
[00274160] Job<shell_6>
```

urbiscript
Session

24.29.3 Slots

- **asJob**
Return [this](#).
- **asString**
The string `Job<name>` where `name` is the name of the job.
- **backtrace**
The current backtrace of the job as a list of [StackFrames](#). Uses [Traceable.backtrace](#).

```
//#push 1 "file.u"
var s = detach(sleep(10))|;
// Leave some time for s to be started.
sleep(1);
assert
{
    s.backtrace[0].asString == "file.u:1.16-24: sleep";
    s.backtrace[1].asString == "file.u:1.16-24: detach";
};
//#pop
```

urbiscript
Session

- **clone**
Cloning a job is impossible since Job is considered as being an atom.
- **dumpState**
Pretty-print the state of the job.

```
//#push 1 "file.u"
var t = detach(sleep(10))|;
// Leave some time for s to be started.
sleep(1);
t.dumpState;
[00004295] *** Job: shell_10
[00004295] ***     State: sleeping
[00004295] ***     Tags:
[00004295] ***         Tag<Lobby_1>
```

urbiscript
Session

```
[00004297] *** Backtrace:  
[00004297] ***     file.u:1.16-24: sleep  
[00004297] ***     file.u:1.16-24: detach  
//#pop
```

- **name**

The name of the job.

urbiscript
Session

```
detach(sleep(10)).name;  
[00004297] "shell_5"
```

- **setSideEffectFree(*value*)**

If value is true, mark the current job as side-effect free. It indicates whether the current state may influence other parts of the system. This is used by the scheduler to choose whether other jobs need scheduling or not. The default value is false.

- **status**

The current status of the job (starting, running, ...), and its properties (frozen, ...).

- **tags**

The list of [Tags](#) that manage this job.

- **terminate**

Kill this job.

urbiscript
Session

```
var r = detach({ sleep(1s); echo("done") })|;  
assert (r in jobs);  
r.terminate;  
assert (r not in jobs);  
sleep(2s);
```

- **timeShift**

Get the total amount of time during which we were frozen.

urbiscript
Session

```
tag: r = detach({ sleep(3); echo("done") })|;  
tag.freeze();  
sleep(2);  
tag.unfreeze();  
Math.round(r.timeShift);  
[00000001] 2
```

- **waitForChanges**

Resume the scheduler, putting the current Job in a waiting status. The scheduler may reschedule the job immediately.

- **waitForTermination**

Wait for the job to terminate before resuming execution of the current one. If the job has already terminated, return immediately.

24.30 Kernel1

This object plays the role of a name-space in which obsolete functions from urbiscript 1.0 are provided for backward compatibility. Do not use these functions, scheduled for removal.

24.30.1 Prototypes

- [Singleton](#)

24.30.2 Construction

Since it is a [Singleton](#), you are not expected to build other instances.

24.30.3 Slots

- **commands**

Ignored for backward compatibility.

- **connections**

Ignored for backward compatibility.

- **copy(*binary*)**

Obsolete syntax for *binary*.copy, see [Binary](#).

```
// copy.
var a = BIN 10;0123456789
[00000001] BIN 10
0123456789

var b = Kernel1.copy(a);
[00000003:warning] !!! 'copy(binary)' is deprecated, use 'binary.copy'
[00000004] BIN 10
0123456789

echo (b);
[00000005] *** BIN 10
0123456789
```

urbiscript
Session

- **devices**

Ignored for backward compatibility.

- **events**

Ignored for backward compatibility.

- **functions**

Ignored for backward compatibility.

- **isvoid(*obj*)**

Obsolete syntax for *obj*.isVoid, see [Object](#).

- **noop**

Do nothing. Use {} instead.

- **ping**

Return time verbosely, see [System](#).

```
Kernel1.ping;
[00000421] *** pong time=0.12s
```

urbiscript
Session

- **reset**

Ignored for backward compatibility.

- **runningcommands**

Ignored for backward compatibility.

- **seq(*number*)**

Obsolete syntax for *number*.seq, see [Float](#).

- **size(*list*)**

Obsolete syntax for *list*.size, see [List](#).

Assertion
Block

```
Kernel1.size([1, 2, 3]) == [1, 2, 3].size;
[00000002:warning] !!! 'size(list)' is deprecated, use 'list.size'
```

- **strict**

Ignored for backward compatibility.

- **strlen(string)**

Obsolete syntax for `string.length`, see [String](#).

Assertion Block

```
Kernel1,strlen("123") == "123".length;
[00000002:warning] !!! 'strlen(string)' is deprecated, use 'string.length'
```

- **taglist**

Ignored for backward compatibility.

- **undefall**

Ignored for backward compatibility.

- **unstrict**

Ignored for backward compatibility.

- **uservars**

Ignored for backward compatibility.

- **vars**

Ignored for backward compatibility.

24.31 Lazy

Lazies are objects that hold a lazy value, that is, a not yet evaluated value. They provide facilities to evaluate their content only once (*memoization*) or several times. Lazy are essentially used in call messages, to represent lazy arguments, as described in [Section 24.4](#).

24.31.1 Examples

24.31.1.1 Evaluating once

One usage of lazy values is to avoid evaluating an expression unless it's actually needed, because it's expensive or has undesired side effects. The listing below presents a situation where an expensive-to-compute value (`heavy_computation`) might be needed zero, one or two times. The objective is to save time by:

- Not evaluating it if it's not needed.
- Evaluating it only once if it's needed once or twice.

We thus make the wanted expression lazy, and use the `value` method to fetch its value when needed.

urbiscript Session

```
// This function supposedly performs expensive computations.
function heavy_computation()
{
    echo("Heavy computation");
    return 1 + 1;
};

// We want to do the heavy computations only if needed,
// and make it a lazy value to be able to evaluate it "on demand".
```

```

var v = Lazy.new(closure () { heavy_computation() });
[00000000] heavy_computation()
/* some code */;
// So far, the value was not needed, and heavy_computation
// was not evaluated.
/* some code */;
// If the value is needed, heavy_computation is evaluated.
v.value;
[00000000] *** Heavy computation
[00000000] 2
// If the value is needed a second time, heavy_computation
// is not reevaluated.
v.value;
[00000000] 2

```

24.31.1.2 Evaluating several times

Evaluating a lazy several times only makes sense with lazy arguments and call messages. See example with call messages in [Section 24.4.1.1](#).

24.31.2 Caching

[Lazy](#) is meant for functions without argument. If you need *caching* for functions that depend on arguments, it is straightforward to implement using a [Dictionary](#). In the future urbiscript might support dictionaries whose indices are not only strings, but in the meanwhile, convert the arguments into strings, as the following sample object demonstrates.

```

class UnaryLazy
{
    function init(f)
    {
        results = [ => ];
        func = f;
    };
    function value(p)
    {
        var sp = p.asString;
        if (results.has(sp))
            return results[sp];
        var res = func(p);
        results[sp] = res |
        res
    };
    var results;
    var func;
} |
// The function to cache.
var inc = function(x) { echo("incing " + x) | x+1 } |
// The function with cache.
// Use "getSlot" to get the function, not its evaluation.
var p = UnaryLazy.new(getSlot("inc"));
[00062847] UnaryLazy_0x78b750
p.value(1);
[00066758] *** incing 1
[00066759] 2
p.value(1);
[00069058] 2
p.value(2);
[00071558] *** incing 2
[00071559] 3
p.value(2);
[00072762] 3
p.value(1);

```

urbiscript
Session

```
[00074562] 2
```

24.31.3 Prototypes

- Comparable

24.31.4 Construction

Lazies are seldom instantiated manually. They are mainly created automatically when a lazy function call is made (see [Section 23.3.4](#)). One can however create a lazy value with the standard `new` method of `Lazy`, giving it an argument-less function which evaluates to the value made lazy.

urbiscript
Session

```
Lazy.new(closure () { /* Value to make lazy */ 0 });
[00000000] 0
```

24.31.5 Slots

- Whether `this` and `that` are the same source code and value (an not yet evaluated Lazy is never equal to an evaluated one).

Assertion
Block

```
Lazy.new(closure () { 1 + 1 }) == Lazy.new(closure () { 1 + 1 });
Lazy.new(closure () { 1 + 2 }) != Lazy.new(closure () { 2 + 1 });
```

urbiscript
Session

```
{
  var l1 = Lazy.new(closure () { 1 + 1 });
  var l2 = Lazy.new(closure () { 1 + 1 });
  assert (l1 == l2);
  l1.eval;
  assert (l1 != l2);
  l2.eval;
  assert (l1 == l2);
};
```

- `asString`

The conversion to `String` of the body of a non-evaluated argument.

Assertion
Block

```
Lazy.new(closure () { echo(1); 1 }).asString == "echo(1);\n1";
```

- `eval`

Force the evaluation of the held lazy value. Two calls to `eval` will systematically evaluate the expression twice, which can be useful to duplicate its side effects.

- `value`

Return the held value, potentially evaluating it before. `value` performs memoization, that is, only the first call will actually evaluate the expression, subsequent calls will return the cached value. Unless you want to explicitly trigger side effects from the expression by evaluating it several time, this should be preferred over `eval` to avoid evaluating the expression several times uselessly.

24.32 List

Lists implement possibly-empty ordered (heterogeneous) collections of objects.

24.32.1 Prototypes

- Container
- RangeIterable
- Orderable

24.32.2 Examples

Since lists are RangeIterable, they also support RangeIterable.all and RangeIterable.any.

```
// Are all elements positive?  
! [-2, 0, 2, 4].all(function (e) { 0 < e });  
// Are all elements even?  
[-2, 0, 2, 4].all(function (e) { e % 2 == 0 });  
  
// Is there any even element?  
! [-3, 1, -1].any(function (e) { e % 2 == 0 });  
// Is there any positive element?  
[-3, 1, -1].any(function (e) { 0 < e });
```

Assertion Block

24.32.3 Construction

Lists can be created with their literal syntax: a possibly empty sequence of expressions in square brackets, separated by commas. Non-empty lists may actually end with a comma.

```
[]; // The empty list  
[00000000] []  
[1, "2", [3,],];  
[00000000] [1, "2", [3]]
```

urbiscript Session

However, new can be used as expected.

```
List.new;  
[00000001] []  
[1, 2, 3].new;  
[00000002] [1, 2, 3]
```

urbiscript Session

24.32.4 Slots

- append(that)
- Deprecated alias for '+='.

```
var one = [1]!;  
one.append(["one", [1]]);  
[00000005:warning] !!! 'list.append(that)' is deprecated, use 'list += that'  
[00000005] [1, "one", [1]]
```

urbiscript Session

- argMax(fun = function(a, b) { a < b })

The index of the (leftmost) “largest” member based on the comparison function *fun*.

```
[1].argMax == 0;  
[1, 2].argMax == 1;  
[1, 2, 2].argMax == 1;  
[2, 1].argMax == 0;  
[2, -1, 3, -4].argMax == 2;  
  
[2, -1, 3, -4].argMax (function (a, b) { a.abs < b.abs }) == 3;
```

Assertion Block

The list cannot be empty.

urbiscript Session

```
[] .argMax;
[00000007:error] !!! argMax: list cannot be empty
```

- **argMin(*fun* = `function(a, b) { a < b }`)**

The index of the (leftmost) “smallest” member based on the comparison function *fun*.

Assertion Block

```
[1] .argMin == 0;
[1, 2] .argMin == 0;
[1, 2, 1] .argMin == 0;
[2, 1] .argMin == 1;
[2, -1, 3, -4] .argMin == 3;

[2, -1, 3, -4].argMin (function (a, b) { a.abs < b.abs }) == 1;
```

The list cannot be empty.

urbiscript Session

```
[] .argMin;
[00000011:error] !!! argMin: list cannot be empty
```

- **asBool**

Whether not empty.

Assertion Block

```
[] .asBool == false;
[1] .asBool == true;
```

- **asList**

Return the target.

urbiscript Session

```
{
  var l = [0, 1, 2];
  assert (l.asList === l);
};
```

- **asString**

A string describing the list. Uses `asPrintable` on its members, so that, for instance, strings are displayed with quotes.

Assertion Block

```
[0, [1], "2"].asString == "[0, [1], \"2\"]";
```

- **back**

The last element of the target. An error if the target is empty.

urbiscript Session

```
assert([0, 1, 2].back == 2);
[] .back;
[00000017:error] !!! back: cannot be applied onto empty list
```

- **clear**

Empty the target.

urbiscript Session

```
var x = [0, 1, 2];
[00000000] [0, 1, 2]
assert(x.clear == []);
```

- **each(*fun*)**

Apply the given functional value *fun* on all members, sequentially.

urbiscript Session

```
[0, 1, 2].each(function (v) {echo (v * v); echo (v * v)});  
[00000000] *** 0  
[00000000] *** 0  
[00000000] *** 1  
[00000000] *** 1  
[00000000] *** 4  
[00000000] *** 4
```

- **eachi(*fun*)**

Apply the given functional value *fun* on all members sequentially, additionally passing the current element index.

```
["a", "b", "c"].eachi(function (v, i) {echo ("%s: %s" % [i, v]));  
[00000000] *** 0: a  
[00000000] *** 1: b  
[00000000] *** 2: c
```

urbiscript
Session

- **'each&'(*fun*)**

Apply the given functional value on all members simultaneously.

```
[0, 1, 2].'each&'(function (v) {echo (v * v); echo (v * v)});  
[00000000] *** 0  
[00000000] *** 1  
[00000000] *** 4  
[00000000] *** 0  
[00000000] *** 1  
[00000000] *** 4
```

urbiscript
Session

- **empty**

Whether the target is empty.

```
[] .empty;  
! [1] .empty;
```

Assertion
Block

- **filter(*fun*)**

The list of all the members of the target that verify the predicate *fun*.

```
do ([0, 1, 2, 3, 4, 5])  
{  
    assert  
    {  
        // Keep only odd numbers.  
        filter(function (v) {v % 2 == 1}) == [1, 3, 5];  
        // Keep all.  
        filter(function (v) { true }) == this;  
        // Keep none.  
        filter(function (v) { false }) == [];  
    };  
};
```

urbiscript
Session

- **foldl(*action*, *value*)**

Fold, also known as *reduce* or *accumulate*, computes a result from a list. Starting from *value* as the initial result, apply repeatedly the binary *action* to the current result and the next member of the list, from left to right. For instance, if *action* were the binary addition and *value* were 0, then folding a list would compute the sum of the list, including for empty lists.

```
[] .foldl(function (a, b) { a + b }, 0) == 0;  
[1, 2, 3].foldl(function (a, b) { a + b }, 0) == 6;  
[1, 2, 3].foldl(function (a, b) { a - b }, 0) == -6;
```

Assertion
Block

- **front**

Return the first element of the target. An error if the target is empty.

urbiscript
Session

```
assert([0, 1, 2].front == 0);
[] .front;
[00000000:error] !!! front: cannot be applied onto empty list
```

- **has(*that*)**

Whether one of the members of the target equals the argument.

Assertion
Block

```
[0, 1, 2].has(1);
! [0, 1, 2].has(5);
```

The infix operators `in` and `not in` use `has` (see [Section 23.1.8.7](#)).

Assertion
Block

```
1 in      [0, 1];
2 not in [0, 1];
!(2 in      [0, 1]);
!(1 not in [0, 1]);
```

- **hash**

Return a `Hash` object corresponding to this list value. Two list hashes are equal if the list have the same size, and elements hashes are equal two by two. See [Object.hash](#).

Assertion
Block

```
[] .hash.isA(Hash);
[] .hash == [] .hash;
[1, "foo"].hash == [1, "foo"].hash;
[0, 1].hash != [1, 0].hash;
```

- **hasSame(*that*)**

Whether one of the members of the target is physically equal to `that`.

urbiscript
Session

```
var y = 1|;
assert
{
  [0, y, 2].hasSame(y);
  ! [0, y, 2].hasSame(1);
};
```

- **head**

Synonym for `front`.

urbiscript
Session

```
assert([0, 1, 2].head == 0);
[] .head;
[00000000:error] !!! head: cannot be applied onto empty list
```

- **insert(*where*, *what*)**

Insert `what` before the value at index `where`, return `this`.

urbiscript
Session

```
{
  var l = [0, 1];
  assert
  {
    l.insert(0, 10) === l;
    l == [10, 0, 1];
    l.insert(2, 20) === l;
    l == [10, 0, 20, 1];
  };
};
```

The index must be valid, to insert past the end, use `insertBack`.

urbiscript
Session

```
[] .insert(0, "foo");
[00044239:error] !!! insert: invalid index: 0
[1, 2, 3].insert(4, 30);
[00044339:error] !!! insert: invalid index: 4
```

- **insertBack(that)**

Insert the given element at the end of the target, return `this`.

```
{
  var l = [0, 1];
  assert
  {
    l.insertBack(2) === l;
    l == [0, 1, 2];
  };
}|;
```

urbiscript
Session

- **insertFront(that)**

Insert the given element at the beginning of the target. Return `this`.

```
{
  var l = [0, 1];
  assert
  {
    l.insertFront(0) === l;
    l == [0, 0, 1];
  };
}|;
```

urbiscript
Session

- **insertUnique(that)**

If `that` is not in `this`, append it. Return `this`.

```
{
  var l = [0, 1];
  assert
  {
    l.insertUnique(0) === l;
    l == [0, 1];
    l.insertUnique(2) === l;
    l == [0, 1, 2];
  };
}|;
```

urbiscript
Session

- **join(sep = "", prefix = "", suffix = "")**

Bounce to `String.join`.

```
["", "ob", ""].join           == "ob";
["", "ob", ""].join("a")      == "aoba";
["", "ob", ""].join("a", "B", "b") == "Baobab";
```

Assertion
Block

- **keys**

The list of valid indexes. This allows uniform iteration over a `Dictionary` or a `List`.

```
{
  var l = ["a", "b", "c"];
  assert
  {
    l.keys == [0, 1, 2];
    {
      var res = [];
      for (var k: l.keys)
```

urbiscript
Session

```

        res << l[k];
    res
}
== 1;
};
}
```

- **map(*fun*)**

Apply the given functional value on every member, and return the list of results.

Assertion Block

```
[0, 1, 2, 3].map(function (v) { v % 2 == 0})
== [true, false, true, false];
```

- **matchAgainst(*handler*, *pattern*)**

If *pattern* is a List of same size, use *handler* to match each member of **this** against the corresponding *pattern*. Return true if the match succeeded, false in other cases.

urbiscript Session

```

assert
{
    ([1, 2] = [1, 2]) == [1, 2];

    ([1, var a] = [1, 2]) == [1, 2];
    a == 2;

    ([var u, var v, var w] = [1, 2, 3]) == [1, 2, 3];
    [u, v, w] == [1, 2, 3];
};

[1, 2] = [2, 1];
[00005863:error] !!! pattern did not match

[1, var a] = [2, 1];
[00005864:error] !!! pattern did not match
[1, var a] = [1];
[00005865:error] !!! pattern did not match
[1, var a] = [1, 2, 3];
[00005865:error] !!! pattern did not match
```

- **max(*fun* = `function(a, b) { a < b }`)**

Return the “largest” member based on the comparison function *fun*.

Assertion Block

```

[1].max == 1;
[1, 2].max == 2;
[2, 1].max == 2;
[2, -1, 3, -4].max == 3;

[2, -1, 3, -4].max (function (a, b) { a.abs < b.abs }) == -4;
```

The list cannot be empty.

urbiscript Session

```
[] .max;
[00000001:error] !!! max: list cannot be empty
```

The members must be comparable.

urbiscript Session

```
[0, 2, "a", 1].max;
[00000002:error] !!! max: argument 2: unexpected "a", expected a Float
```

- **min(*fun* = `function(a, b) { a < b }`)**

Return the “smallest” member based on the comparison function *fun*.

Assertion Block

```
[1].min == 1;
[1, 2].min == 1;
[2, 1].min == 1;
[2, -1, 3, -4].min == -4;

[2, -1, 3, -4].min (function (a, b) { a.abs < b.abs }) == -1;
```

The list cannot be empty.

```
[] .min;
[00000001:error] !!! min: list cannot be empty
```

urbiscript
Session

- **range(*begin, end = nil*)**

Return a sub-range of the list, from the first index included to the second index excluded.
An error if out of bounds. Negative indices are valid, and number from the end.

If *end* is *nil*, calling `range(n)` is equivalent to calling `range(0, n)`.

```
do ([0, 1, 2, 3])
{
    assert
    {
        range(0, 0) == [];
        range(0, 1) == [0];
        range(1) == [0];
        range(1, 3) == [1, 2];

        range(-3, -2) == [1];
        range(-3, -1) == [1, 2];
        range(-3, 0) == [1, 2, 3];
        range(-3, 1) == [1, 2, 3, 0];
        range(-4, 4) == [0, 1, 2, 3, 0, 1, 2, 3];
    };
}!;
[] .range(1, 3);
[00428697:error] !!! range: invalid index: 1
```

urbiscript
Session

- **remove(*val*)**

Remove all elements from the target that are equal to *val*, return `this`.

```
var c = [0, 1, 0, 2, 0, 3];
assert
{
    c.remove(0) === c;    c == [1, 2, 3];
    c.remove(42) === c;  c == [1, 2, 3];
};
```

urbiscript
Session

- **removeBack**

Remove and return the last element of the target. An error if the target is empty.

```
var t = [0, 1, 2];
[00000000] [0, 1, 2]
assert(t.removeBack == 2);
assert(t == [0, 1]);
[] .removeBack;
[00000000:error] !!! removeBack: cannot be applied onto empty list
```

urbiscript
Session

- **removeById(*that*)**

Remove all elements from the target that physically equals *that*.

urbiscript
Session

```
var d = 1|;
var e = [0, 1, d, 1, 2]|;
assert
{
  e.removeById(d) == [0, 1, 1, 2];
  e == [0, 1, 1, 2];
};
```

- **removeFront**

Remove and return the first element from the target. An error if the target is empty.

urbiscript
Session

```
var g = [0, 1, 2]|;
assert
{
  g.removeFront == 0;
  g == [1, 2];
};
[] .removeFront;
[00000000:error] !!! removeFront: cannot be applied onto empty list
```

- **reverse**

Return the target with the order of elements inverted.

Assertion
Block

```
[0, 1, 2].reverse == [2, 1, 0];
```

- **size**

Return the number of elements in the target.

Assertion
Block

```
[0, 1, 2].size == 3;
[] .size == 0;
```

- **sort(*fun* = `function(a, b) { a < b }`)**

A new List with the contents of `this`, sorted with respect to the `comp` comparison function.

urbiscript
Session

```
{|
  var l = [3, 0, -2, 1];
  assert
  {
    l.sort == [-2, 0, 1, 3];
    l      == [3, 0, -2, 1];

    l.sort(function(a, b) {a.abs < b.abs})
           == [0, 1, -2, 3];
  };
};
```

- **subset(*that*)**

Whether the members of `this` are members of `that`.

Assertion
Block

```
[] .subset([]);
[] .subset([1, 2, 3]);
[3, 2, 1].subset([1, 2, 3]);
[1, 3].subset([1, 2, 3]);
[1, 1].subset([1, 2, 3]);

  ! [3].subset([]);
  ! [3, 2].subset([1, 2]);
  ! [1, 2, 3].subset([1, 2]);
```

- **tail**

Return the target, minus the first element. An error if the target is empty.

```
assert([0, 1, 2].tail == [1, 2]);
[] .tail;
[00000000:error] !!! tail: cannot be applied onto empty list
```

urbiscript
Session

- **zip(*fun*, *other*)**

Zip *this* list and the *other* list with the *fun* function, and return the list of results.

```
[1, 2, 3].zip(closure (x, y) { (x, y) }, [4, 5, 6])
    == [(1, 4), (2, 5), (3, 6)];
[1, 2, 3].zip(closure (x, y) { x + y }, [4, 5, 6])
    == [5, 7, 9];
```

Assertion
Block

- **'=='(*that*)**

Check whether all elements in the target and *that*, are equal two by two.

```
[0, 1, 2] == [0, 1, 2];
!([0, 1, 2] == [0, 0, 2]);
```

Assertion
Block

- **'[]'(*n*)**

Return the *n*th member of the target (indexing is zero-based). If *n* is negative, start from the end. An error if out of bounds.

```
assert
{
    ["0", "1", "2"][0] == "0";
    ["0", "1", "2"][2] == "2";
};

["0", "1", "2"][3];
[00007061:error] !!! []: invalid index: 3

assert
{
    ["0", "1", "2"][-1] == "2";
    ["0", "1", "2"][-3] == "0";
};
["0", "1", "2"][-4];
[00007061:error] !!! []: invalid index: -4
```

urbiscript
Session

- **'[]='(*index*, *value*)**

Assign *value* to the element of the target at the given *index*.

```
var f = [0, 1, 2];
[00000000] [0, 1, 2]
assert
{
    (f[1] = 42) == 42;
    f == [0, 42, 2];
};

for (var i: [0, 1, 2])
    f[i] = 10 * f[i];
assert (f == [0, 420, 20]);
```

urbiscript
Session

- **'*'(*n*)**

Return the target, concatenated *n* times to itself.

```
[0, 1] * 0 == [];
[0, 1] * 3 == [0, 1, 0, 1, 0, 1];
```

Assertion
Block

n must be a non-negative integer.

urbiscript
Session

```
[0, 1] * -2;
[00000063:error] !!! *: argument 1: expected non-negative integer, got -2
```

Note that since it is the very same list which is repeatedly concatenated (the content is not cloned), side-effects on one item will reflect on “all the items”.

urbiscript
Session

```
var l = [[]] * 3;
[] [0, 1, 1, 1]
l[0] << 1;
[] [1, 1, 1]
l;
[] [[1], [1], [1]]
```

- **'+' (*other*)**

Return the concatenation of the target and the *other* list.

Assertion
Block

```
[0, 1] + [2, 3] == [0, 1, 2, 3];
[] + [2, 3] == [2, 3];
[0, 1] + [] == [0, 1];
[] + [] == [];
```

The target is left unmodified (contrary to **'+='**).

urbiscript
Session

```
{
  var l = [1, 2, 3];
  assert
  {
    l + 1 == [1, 2, 3, 1, 2, 3];
    l      == [1, 2, 3];
  };
}
```

- **'+=' (*that*)**

Concatenate the contents of the List *that* to *this*, and return *this*. This function modifies its target, contrary to **'+'**. See also **'<<'**.

urbiscript
Session

```
{
  var l = [];
  var alias = l;
  assert
  {
    (l += [1, 2]) == l;
    l == [1, 2];
    (l += [3, 4]) == l;
    l == [1, 2, 3, 4];
    alias == [1, 2, 3, 4];
  };
}
```

- **'-' (*other*)**

Return the target without the elements that are equal to any element in the *other* list.

Assertion
Block

```
[0, 1, 0, 2, 3] - [1, 2] == [0, 0, 3];
[0, 1, 0, 1, 0] - [1, 2] == [0, 0, 0];
```

- **'<<' (*that*)**

A synonym for `insertBack`.

- '`<`' (*other*)

Return whether `this` is less than the *other* list. This is the lexicographic comparison: `this` is “less than” if one of its member is “less than” the corresponding member of *other*:

```
[0, 0, 0] < [0, 0, 1];
[0, 1, 2] < [0, 2, 1];

!([0, 1, 2] < [0, 1, 2]);
!([0, 1, 2] < [0, 0, 2]);
```

Assertion Block

or *other* is a prefix (strict) of `this`:

```
[] < [0];
[] < [0, 1];
[] < [0, 1, 2];
!([0, 1, 2] < [0, 1]);
!([0, 1, 2] < [0, 1, 2]);
```

Assertion Block

Since List derives from `Orderable`, the other order-based operators are defined.

```
[] <= [];
[] <= [0, 1, 2];
[0, 1, 2] <= [0, 1, 2];

[] >= [];
[0, 1, 2] >= [];
[0, 1, 2] >= [0, 1, 2];
[0, 1, 2] >= [0, 0, 2];

!([0] > []);
[0, 1, 2] > [];
!([0, 1, 2] > [0, 1, 2]);
[0, 1, 2] > [0, 0, 2];
```

Assertion Block

24.33 Loadable

Loadable objects can be switched on and off — typically physical devices.

24.33.1 Prototypes

- `Object`

24.33.2 Example

The intended use is rather as follows:

```
class Motor: Loadable
{
    var val = 0;
    function go(var d)
    {
        if (load)
            val += d
        else
            echo("cannot advance, the motor is off")!;
    };
};

[00000002] Motor

var m = Motor.new;
[00000003] Motor_OxADDR

m.load;
[00000004] false
```

urbiscript Session

```
m.go(1);
[00000006] *** cannot advance, the motor is off

m.on;
[00000007] Motor_OxADDR

m.go(123);
m.val;
[00000009] 123
```

24.33.3 Construction

`Loadable` can be constructed, but it hardly makes sense. This object should serve as a prototype.

24.33.4 Slots

- `load`

The current status.

- `off(val)`

Set `load` to `false` and return `this`.

Assertion Block

```
do (Loadable.new)
{
    assert
    {
        !load;
        off === this;
        !load;
        on === this;
        load;
        off === this;
        !load;
    };
}
```

- `on(val)`

Set `load` to `true` and return `this`.

Assertion Block

```
do (Loadable.new)
{
    assert
    {
        !load;
        on === this;
        load;
        on === this;
        load;
    };
}
```

- `toggle`

Set `load` from `true` to `false`, and vice-versa. Return `val`.

Assertion Block

```
do (Loadable.new)
{
    assert
    {
        !load;
        toggle === this;
        load;
    };
}
```

```

    toggle === this;
    !load;
};

};


```

24.34 Lobby

A *lobby* is the local environment for each (remote or local) connection to an Urbi server.

24.34.1 Prototypes

- `Channel topLevel`, an instance of `Channel` with an empty Channel name.

24.34.2 Construction

A lobby is implicitly created at each connection. At the top level, `this` is a *Lobby*.

```

this.protos;
[00000001] [Lobby]
this.protos[0].protos;
[00000003] [Channel_OxADDR]


```

urbiscript
Session

Lobbies cannot be cloned, they must be created using `create`.

```

Lobby.new;
[00000177:error] !!! new: 'Lobby' objects cannot be cloned
Lobby.create;
[00000174] Lobby_0x126450


```

urbiscript
Session

24.34.3 Examples

Since every lobby is-a `Channel`, one can use the methods of `Channel`.

```

lobby << 123;
[00478679] 123
lobby << "foo";
[00478679] "foo"


```

urbiscript
Session

24.34.4 Slots

- `authors`

Credit the authors of Urbi SDK. See also `thanks` and [Section 1.4](#).

```

lobby.authors;
[00478679] *** Authors of Urbi:
[00478679] *** - Jean-Christophe Baillie (original designer)
[00478679] ***
[00478679] *** Urbi 2 and subsequent versions:
[00478679] *** - Akim Demaille
[00478679] *** - Matthieu Nottale
[00478679] *** - Quentin Hocquet
[00478679] ***
[00478679] *** See also 'thanks;'.


```

urbiscript
Session

- `banner`

Internal. Display Urbi SDK banner.

urbiscript
Session

```

lobby.banner;
[00006124] *** ****
[00006124] *** Urbi version 2.7.4 rev. 268868e
[00006124] *** Copyright (C) 2004-2012 Gostai S.A.S.
[00006124] ***
[00006124] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00006124] *** be used under certain conditions. Type 'license;',
[00006124] *** 'authors;', or 'copyright,' for more information.
[00006124] ***
[00006124] *** Check our community site: http://www.urbiforge.org.
[00006124] *** ****

```

- **bytesReceived**

The number of bytes that were “input” to `this`. See also `receive`.

urbiscript
Session

```

var l = Lobby.create();
assert (l.bytesReceived == 0);

l.receive("123456789");
[00000022] 123456789
assert (l.bytesReceived == 10);

l.receive("1234");
[00000023] 1234
assert (l.bytesReceived == 15);

```

- **bytesSent**

The number of bytes that were “output” by `this`. See also `send` and `write`.

urbiscript
Session

```

var l = Lobby.create();
assert (l.bytesSent == 0);

l.send("0123456789");
[00011988] 0123456789
// 22 = "[00011988] 0123456789\n".size.
assert (l.bytesSent == 22);

l.send("xx", "label");
[00061783:label] xx
// 20 = "[00061783:label] xx\n".size.
assert (l.bytesSent == 42);

l.write("[01234567]\n");
[01234567]
assert (l.bytesSent == 53);

```

- **connected**

Whether `this` is connected.

Assertion
Block

```
connected;
```

- **connectionTag**

The tag of all code executed in the context of `this`. This tag applies to `this`, but the top-level loop is immune to `Tag.stop`, therefore `connectionTag` controls every thing that was launched from this lobby, yet the lobby itself is still usable.

urbiscript
Session

```

every (1s) echo(1), sleep(0.5s); every (1s) echo(2),
sleep(1.2s);
connectionTag.stop;
[00000507] *** 1
[00001008] *** 2

```

```
[00001507] *** 1
[00002008] *** 2

"We are alive!";
[00002008] "We are alive!"

every (1s) echo(3), sleep(0.5s); every (1s) echo(4),
sleep(1.2s);
connectionTag.stop;
[00003208] *** 3
[00003710] *** 4
[00004208] *** 3
[00004710] *** 4

"and kicking!";
[00002008] "and kicking!"
```

Of course, a background job may stop a foreground one.

```
{ sleep(1.2s); connectionTag.stop; },
// Note the ';', this is a foreground statement.
every (1s) echo(5);
[00005008] *** 5
[00005508] *** 5

"bye!";
[00006008] "bye!"
```

urbiscript
Session

- **copyright(*deep* = true)**

Display the copyright of Urbi SDK. Include copyright information about sub-components if *deep*.

```
lobby.copyright(false);
[00028588] *** Urbi version 2.7.4 rev. 268868e
[00028588] *** Copyright (C) 2004-2012 Gostai S.A.S.

lobby.copyright;
[00041874] *** Urbi version 2.7.4 rev. 268868e
[00041874] *** Copyright (C) 2004-2012 Gostai S.A.S.
[00041874] ***
[00041874] *** Libport version 2.7.4 rev. 268868e
[00041874] *** Copyright (C) 2004-2012 Gostai S.A.S.
```

urbiscript
Session

- **create**

Instantiate a new Lobby.

```
Lobby.create.isA(Lobby);
```

Assertion
Block

- **echo(*value*, *channel* = "")**

Send *value*.asString to **this**, prefixed by the **String** *channel* name if specified. This is the preferred way to send informative messages (prefixed with '***').

```
lobby.echo("111", "foo");
[00015895:foo] *** 111
lobby.echo(222, "");
[00051909] *** 222
lobby.echo(333);
[00055205] *** 333
```

urbiscript
Session

- **echoEach(*list*, *channel* = "")**

Apply **echo(*m*, *channel*)** for each member *m* of *list*.

urbiscript
Session

```
lobby.echo([1, "2"], "foo");
[00015895:foo] *** [1, "2"]

lobby.echoEach([1, "2"], "foo");
[00015895:foo] *** 1
[00015895:foo] *** 2

lobby.echoEach([], "foo");
```

- **instances**

A list of the currently alive lobbies. It contains at least the Lobby object itself, and the current `lobby`.

Assertion Block

```
lobby in Lobby.instances;
Lobby in Lobby.instances;
```

- **license**

Display the end user license agreement of the Urbi SDK.

urbiscript Session

```
lobby.license;
[00000000] ***
[00000000] ***
[00000000] ***
[00000000] *** GNU AFFERO GENERAL PUBLIC LICENSE
[00000000] *** Version 3, 19 November 2007
[00000000] ***
[00000000] *** Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
[00000000] *** Everyone is permitted to copy and distribute verbatim copies
[00000000] *** of this license document, but changing it is not allowed.
[00000000] ***
[...]
[00000000] *** For more information on this, and how to apply and follow the GNU AGPL, see
[00000000] *** <http://www.gnu.org/licenses/>.
```

- **lobby**

The lobby of the current connection. This is typically `this`.

Assertion Block

```
Lobby.lobby === this;
```

But when several connections are active (e.g., when there are remote connections), it can be different from the target of the call.

urbiscript Session

```
Lobby.create| Lobby.create|;
for (var l : lobbies)
    assert (l.lobby == Lobby.lobby);
```

- **onDisconnect(lobby)**

Event launched when `this` is disconnected. There is a single event instance for all the lobby, `Lobby.onDisconnect`, the disconnected lobby being passed as argument.

- **quit**

Shut this lobby down, i.e., close the connection. The server is still running, see `System.shutdown` to quit the server.

- **receive(value)**

This is low-level routine. Pretend the `String value` was received from the connection. There is no guarantee that `value` will be the next program block that will be processed: for instance, if you load a file which, in its middle, uses `lobby.receive("foo")`, then `"foo"` will be appended after the end of the file.

urbiscript Session

```
Lobby.create.receive("12;");
[00478679] 12
```

- `remoteIP`

When `this` is connected to a remote server, it's Internet address.

- `send(value, channel = "")`

This is low-level routine. Send the `String value` to `this`, prefixed by the `String channel` name if specified.

```
lobby.send("111", "foo");
[00015895:foo] 111
lobby.send("222", "");
[00051909] 222
lobby.send("333");
[00055205] 333
```

urbiscript
Session

- `thanks`

Credit the contributors of Urbi SDK. See also `authors` and [Section 1.4](#).

```
lobby.thanks;
[00478679] *** Contributors to Urbi:
[00478679] ***
[00478679] *** - Adam Oleksy <adam.oleksy@pwr.wroc.pl>
[00478679] *** - Alexandre Morgand
[...]
[00478679] ***
[00478679] *** See also 'authors;'.
```

urbiscript
Session

- `wall(value, channel = "")`

Perform `echo(value, channel)` on all the existing lobbies (except Lobby itself).

```
Lobby.wall("111", "foo");
[00015895:foo] *** 111
```

urbiscript
Session

- `write(value)`

This is low-level routine. Send the `String value` to the connection. Note that because of buffering, the output might not be visible before an end-of-line character is output.

```
lobby.write("[");
lobby.write("999999999:");
lobby.write("myTag] ");
lobby.write("Hello, World!");
lobby.write("\n");
[999999999:myTag] Hello, World!
```

urbiscript
Session

24.35 Location

This class aggregates two Positions and provides a way to print them as done in error messages.

24.35.1 Prototypes

- `Object`

24.35.2 Construction

Without argument, a newly constructed Location has its Positions initialized to the first line and the first column.

```
Location.new;
[00000001] 1.1
```

urbiscript
Session

With a Position argument *p*, the Location will clone the Position into the begin and end Positions.

urbiscript
Session

```
Location.new(Position.new("file.u",14,25));
[00000001] file.u:14.25
```

With two Positions arguments *begin* and *end*, the Location will clone both Positions into its own fields.

urbiscript
Session

```
Location.new(Position.new("file.u",14,25), Position.new("file.u",14,35));
[00000001] file.u:14.25-34
```

24.35.3 Slots

- `'==' (other)`

Compare the begin and end Position.

urbiscript
Session

```
{
  var p1 = Position.new("file.u",14,25);
  var p2 = Position.new("file.u",16,35);
  var p3 = Position.new("file.u",18,45);
  assert
  {
    Location.new(p1, p3) != Location.new(p1, p2);
    Location.new(p1, p3) == Location.new(p1, p3);
    Location.new(p1, p3) != Location.new(p2, p3);
  };
}
```

- `asString`

Present Locations with less variability as possible as either:

- `'file:ll.cc'`
- `'file:ll.cc-cc'`
- `'file:ll.cc-ll.cc'`

or the same without file name when the file name is not defined.

Assertion
Block

```
Location.new(Position.new("file.u",14,25)).asString == "file.u:14.25";
Location.new(Position.new(14,25)).asString == "14.25";

Location.new(
  Position.new("file.u", 14, 25),
  Position.new("file.u", 14, 35)
).asString == "file.u:14.25-34";

Location.new(
  Position.new(14, 25),
  Position.new(14, 35)
).asString == "14.25-34";

Location.new(
  Position.new("file.u", 14, 25),
  Position.new("file.u", 15, 35)
).asString == "file.u:14.25-15.34";

Location.new(
  Position.new(14, 25),
  Position.new(15, 35)
).asString == "14.25-15.34";
```

- **begin**

The begin Position used by the Location. Modifying a copy of this field does not modify the Location.

```
Location.new(
    Position.new("file.u", 14, 25),
    Position.new("file.u", 16, 35)
).begin == Position.new("file.u", 14, 25);
```

Assertion Block

- **end**

The end Position used by the Location. Modifying a copy of this field does not modify the Location.

```
Location.new(
    Position.new("file.u",14,25),
    Position.new("file.u",16,35)
).end == Position.new("file.u",16,35);
```

Assertion Block

24.36 Logger

Logger is used to report information to the final user or to the developer. It allows to pretty print warnings, errors, debug messages or simple logs. Logger can also be used as Tag objects for it to handle nested calls indentation. A log message is assigned a category which is shown between brackets at beginning of lines, and a level which defines the context in which it has to be shown (see [Section 22.1.2](#)). Log level definition and categories filtering can be changed using environment variables defined in [Section 22.1.2](#).

24.36.1 Examples

The proper use of Loggers is to instantiate your own category, and then to use the operator << for log messages, possibly qualified by the proper level (in increase order of importance: **dump**, **debug**, **trace**, **log**, **warn**, **err**):

```
var logger = Logger.new("Category")|;
logger.dump << "Low level debug message"|;
// Nothing displayed, unless the debug level is set to DUMP.

logger.warn << "something wrong happened, proceeding"|;
[      Category      ] something wrong happened, proceeding

logger.err << "something really bad happened!"|;
[      Category      ] something really bad happened!
```

urbiscript Session

You may also directly use the logger and passing arguments to these slots.

```
Logger.log("message", "Category") |;
[      Category      ] message

Logger.log("message", "Category") :
{
  Logger.log("indented message", "SubCategory")
}|;
[      Category      ] message
[      SubCategory   ] indented message
```

urbiscript Session

24.36.2 Prototypes

- **Tag**

24.36.3 Construction

Logger can be used as is, without being cloned. It is possible to clone Logger defining a category and/or a level of debug.

urbiscript
Session

```
Logger.log("foo");
[      Logger      ] foo
[00004702] Logger<Logger>

Logger.log("foo", "Category") |;
[      Category     ] foo

var l = Logger.new("Category2");
[00004703] Logger<Category2>

l.log("foo") |;
[      Category2    ] foo

l.log("foo", "ForcedCategory") |;
[      ForcedCategory ] foo
```

24.36.4 Slots

- `'<<'(object)`

Allow to use the Logger object as a [Channel](#). This slot can only be used if both category and level have been defined when cloning.

urbiscript
Session

```
l = Logger.new("Category", Logger.Levels.Log);
[00090939] Logger<Category>
l << "foo";
[      Category      ] foo
[00091939] Logger<Category>
```

- `debug(message = "", category = category)`

Report a debug *message* of *category* to the user. It will be shown if the debug level is Debug or Dump. Return [this](#) to allow chained operations.

urbiscript
Session

```
// None of these are displayed unless the current level is at least DEBUG.
logger.debug << "debug 1"|;
logger.debug("debug 2")|;
logger.debug("debug 3", "Category2")|;
```

- `dump(message = "", category = category)`

Report a debug *message* of *category* to the user. It will be shown if the debug level is Dump. Return [this](#) to allow chained operations.

urbiscript
Session

```
// None of these are displayed unless the current level is at least DEBUG.
logger.dump << "dump 1"|;
logger.dump("dump 2")|;
logger.dump("dump 3", "Category2")|;
```

- `err(message = "", category = category)`

Report an error *message* of *category* to the user. Return [this](#) to allow chained operations.

- `init(category)`

Define the *category* of the new Logger object. If no category is given the new Logger will inherit the category of its prototype.

- `Levels`

An [Enumeration](#) of the log levels defined in [Section 22.1.2](#).

urbiscript
Session

```
Logger.Levels.values;
[00000001] [None, Log, Trace, Debug, Dump]
```

- **log(*message* = "", *category* = category)**

Report a debug *message* of *category* to the user. It will be shown if debug is not disabled. Return `this` to allow chained operations.

```
logger.log << "log 1"|;
[     Category      ] log 1

logger.log("log 2")|;
[     Category      ] log 2

logger.log("log 3", "Category2")|;
[     Category2     ] log 3
```

urbiscript
Session

- **onEnter**

The primitive called when `Logger` is used as a Tag and is entered. This primitive only increments the indentation level.

- **onLeave**

The primitive called when `Logger` is used as a Tag and is left. This primitive only decrements the indentation level.

- **trace(*message* = "", *category* = category)**

Report a debug *message* of *category* to the user. It will be shown if the debug level is Trace, Debug or Dump. Return `this` to allow chained operations.

```
// None of these are displayed unless the current level is at least TRACE.
logger.trace << "trace 1"|;
logger.trace("trace 2")|;
logger.trace("trace 3", "Category2")|;
```

urbiscript
Session

- **warn(*message* = "", *category* = category)**

Report a warning *message* of *category* to the user. Return `this` to allow chained operations.

```
logger.warn << "warn 1"|;
[     Category      ] warn 1

logger.warn("warn 2")|;
[     Category      ] warn 2

logger.warn("warn 3", "Category2")|;
[     Category2     ] warn 3
```

urbiscript
Session

24.37 Math

This object is actually meant to play the role of a name-space in which the mathematical functions are defined with a more conventional notation. Indeed, in an object-oriented language, writing `pi.cos` makes perfect sense, yet `cos(pi)` is more usual.

24.37.1 Prototypes

- **Singleton**

24.37.2 Construction

Since it is a [Singleton](#), you are not expected to build other instances.

24.37.3 Slots

- `abs(float)`

Bounce to `float.abs`.

Assertion Block

```
Math.abs(1) == 1;
Math.abs(-1) == 1;
Math.abs(0) == 0;
Math.abs(3.5) == 3.5;
```

- `acos(float)`

Bounce to `float.acos`.

- `asin(float)`

Bounce to `float.asin`.

- `atan(float)`

Bounce to `float.atan`.

Assertion Block

```
Math.atan(1) ~= pi/4;
```

- `atan2(x, y)`

Bounce to `x.atan2(y)`.

Assertion Block

```
Math.atan2(2, 2) ~= pi/4;
Math.atan2(-2, 2) ~= -pi/4;
```

- `cos(float)`

Bounce to `float.cos`.

Assertion Block

```
Math.cos(0) == 1;
Math.cos(pi) ~= -1;
```

- `exp(float)`

Bounce to `float.exp`.

- `inf`

Bounce to [Float.inf](#).

- `log(float)`

Bounce to `float.log`.

Assertion Block

```
Math.log(1) == 0;
```

- `max(arg1, ...)`

Bounce to `[arg1, ...].max`, see [List.max](#).

Assertion Block

```
max( 100, 20, 3 ) == 100;
max("100", "20", "3") == "3";
```

- `min(arg1, ...)`

Bounce to `[arg1, ...].min`, see [List.min](#).

Assertion Block

```
min( 100, 20, 3 ) == 3;
min("100", "20", "3") == "3";
```

- **nan**
Bounce to `Float.nan`.
- **pi**
Bounce to `Float.pi`.
- **random(*float*)**
Bounce to `float.random`.
- **round(*float*)**
Bounce to `float.round`.

```
Math.round(1) == 1;
Math.round(1.1) == 1;
Math.round(1.49) == 1;
Math.round(1.5) == 2;
Math.round(1.51) == 2;
```

Assertion Block

- **sign(*float*)**
Bounce to `float.sign`.

```
Math.sign(2) == 1;
Math.sign(-2) == -1;
Math.sign(0) == 0;
```

Assertion Block

- **sin(*float*)**
Bounce to `float.sin`.

```
Math.sin(0) == 0;
Math.sin(pi) ~= 0;
```

Assertion Block

- **sqr(*float*)**
Bounce to `float.sqr`.

```
Math.sqr(2.2) ~= 4.84;
```

Assertion Block

- **sqrt(*float*)**
Bounce to `float.sqrt`.

```
Math.sqrt(4) == 2;
```

Assertion Block

- **srandom(*float*)**
Bounce to `float.srandom`.

- **tan(*float*)**
Bounce to `float.tan`.

```
Math.tan(pi/4) ~= 1;
```

Assertion Block

- **trunc(*float*)**
Bounce to `float.trunc`.

```
Math.trunc(1) == 1;
Math.trunc(1.1) == 1;
Math.trunc(1.49) == 1;
Math.trunc(1.5) == 1;
Math.trunc(1.51) == 1;
```

Assertion Block

24.38 Mutex

Mutex allow to define critical sections.

24.38.1 Prototypes

- [Tag](#)

24.38.2 Construction

A Mutex can be constructed like any other Tag but without name.

urbiscript
Session

```
var m = Mutex.new;
[00000000] Mutex_0x964ed40
```

You can define critical sections by tagging your code using the Mutex.

urbiscript
Session

```
var m = Mutex.new();
m: echo("this is critical section");
[00000001] *** this is critical section
```

As a critical section, two pieces of code tagged by the same “Mutex” will never be executed at the same time.

24.38.3 Slots

- [asMutex](#)

Return [this](#).

urbiscript
Session

```
var m1 = Mutex.new();
assert
{
    m1.asMutex === m1;
};
```

24.39 nil

The special entity `nil` is an object used to denote an empty value. Contrary to `void`, it is a regular value which can be read.

24.39.1 Prototypes

- [Singleton](#)

24.39.2 Construction

Being a singleton, `nil` is not to be constructed, just used.

Assertion
Block

```
nil == nil;
```

24.39.3 Slots

- [isNil](#)

Whether [this](#) is `nil`. I.e., true. See also `Object.isNil`.

Assertion
Block

```
nil.isNil;
!Object.isNil;
```

24.40 Object

`Object` includes the mandatory primitives for all objects in urbiscript. All objects in urbiscript must inherit (directly or indirectly) from it.

24.40.1 Prototypes

- `Comparable`
- `Global`

24.40.2 Construction

A fresh object can be instantiated by cloning `Object` itself.

```
Object.new;
[00000421] Object_0x00000000
```

urbiscript
Session

The keyword `class` also allows to define objects intended to serve as prototype of a family of objects, similarly to classes in traditional object-oriented programming languages (see Section 12.4).

```
{
  class Foo
  {
    var attr = 23;
  };
  assert
  {
    Foo.localSlotNames == ["asFoo", "attr", "type"];
    Foo.asFoo === Foo;
    Foo.attr == 23;
    Foo.type == "Foo";
  };
}
```

urbiscript
Session

24.40.3 Slots

- `acceptVoid`

Return `this`. See `void` to know why.

```
{
  var o = Object.new;
  assert(o.acceptVoid === o);
};
```

urbiscript
Session

- `addProto(proto)`

Add `proto` into the list of prototypes of `this`. Return `this`.

```
do (Object.new)
{
  assert
  {
    addProto(Orderable) === this;
    protos == [Orderable, Object];
  };
}|;
```

urbiscript
Session

- `allProto`

A list with `this`, its parents, their parents,...

Assertion
Block

```
123.allProtos.size == 12;
```

- **allSlotNames**

Deprecated alias for `slotNames`.

Assertion Block

```
Object.allSlotNames == Object.slotNames;
```

- **apply(args)**

“Invoke `this`”. The size of the argument list, `args`, must be one. This argument is ignored. This function exists for compatibility with `Code.apply`.

Assertion Block

```
Object.apply([this]) === Object;
Object.apply([1]) === Object;
```

- **as(type)**

Convert `this` to `type`. This is syntactic sugar for `asType` when `Type` is the type of `type`.

Assertion Block

```
12.as(Float) == 12;
"12".as(Float) == 12;
12.as(String) == "12";
Object.as(Object) === Object;
```

- **asBool**

Whether `this` is “true”, see [Section 24.3.3](#).

urbiscript Session

```
assert
{
    Global.asBool == true;
    nil.asBool == false;
};

void.asBool;
[00000421:error] !!! unexpected void
```

- **bounce(name)**

Return `this.name` transformed from a method into a function that takes its target (its “`this`”) as first and only argument. `this.name` must take no argument.

Assertion Block

```
{ var myCos = Object.bounce("cos"); myCos(0) } == 0.cos;
{ var myType = bounce("type"); myType(Object); } == "Object";
{ var myType = bounce("type"); myType(3.14); } == "Float";
```

- **callMessage(msg)**

Invoke the `CallMessage msg` on this.

- **clone**

Clone `this`, i.e., create a fresh, empty, object, which sole prototype is `this`.

Assertion Block

```
Object.clone.protos == [Object];
Object.clone.localSlotNames == [];
```

- **cloneSlot(from, to)**

Set the new slot `to` using a clone of `from`. This can only be used into the same object.

urbiscript Session

```
var foo = Object.new|;
cloneSlot("foo", "bar") |;
assert(!(foo === bar));
```

- `copySlot(from, to)`

Same as `cloneSlot`, but the slot aren't cloned, so the two slot are the same.

```
var moo = Object.new|;
cloneSlot("moo", "loo")|;
assert(!(moo === loo));
```

urbiscript
Session

- `createSlot(name)`

Create an empty slot (which actually means it is bound to `void`) named *name*. Raise an error if *name* was already defined.

```
do (Object.new)
{
    assert(!hasLocalSlot("foo"));
    assert(createSlot("foo").isVoid);
    assert(hasLocalSlot("foo"));
}|;
```

urbiscript
Session

- `dump(depth)`

Describe `this`: its prototypes and slots. The argument *depth* specifies how recursive the description is: the greater, the more detailed. This method is mostly useful for debugging low-level issues, for a more human-readable interface, see also `inspect`.

```
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.dump(0);
[00015137] *** Float_0x240550 {
[00015137] ***     /* Special slots */
[00015137] ***     protos = Float
[00015137] ***     value = 2
[00015137] ***     /* Slots */
[00015137] ***     attr = String_0x23a750 <...>
[00015137] ***         /* Properties */
[00015137] ***         prop = String_0x23a7a0 <...>
[00015137] ***     }
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.dump(1);
[00020505] *** Float_0x240550 {
[00020505] ***     /* Special slots */
[00020505] ***     protos = Float
[00020505] ***     value = 2
[00020505] ***     /* Slots */
[00020505] ***     attr = String_0x23a750 {
[00020505] ***         /* Special slots */
[00020505] ***         protos = String
[00020505] ***         /* Slots */
[00020505] ***     }
[00020505] ***     /* Properties */
[00020505] ***     prop = String_0x239330 {
[00020505] ***         /* Special slots */
[00020505] ***         protos = String
[00020505] ***         /* Slots */
[00020505] ***     }
[00020505] *** }
```

urbiscript
Session

- `getPeriod`

Deprecated. Use `System.period` instead.

- `getProperty(slotName, propName)`

The value of the *propName* property associated to the slot *slotName* if defined. Raise an error otherwise.

```
const var myPi = 3.14|;
assert (getProperty("myPi", "constant"));
```

urbiscript
Session

```
getProperty("myPi", "foobar");
[00000045:error] !!! property lookup failed: myPi->foobar
```

- **getLocalSlot(*name*)**

The value associated to *name* in **this**, excluding its ancestors (contrary to **getSlot**).

urbiscript
Session

```
var a = Object.new();
// Local slot.
var a.slot = 21;
assert
{
  a.locateSlot("slot") === a;
  a.getLocalSlot("slot") == 21;
};

// Inherited slot are not looked-up.
assert { a.locateSlot("init") == Object };
a.getSlot("init");
[00041066:error] !!! lookup failed: init
```

- **getSlot(*name*)**

The value associated to *name* in **this**, possibly after a look-up in its prototypes (contrary to **getLocalSlot**).

urbiscript
Session

```
var b = Object.new();
var b.slot = 21;

assert
{
  // Local slot.
  b.locateSlot("slot") === b;
  b.getSlot("slot") == 21;

  // Inherited slot.
  b.locateSlot("init") === Object;
  b.getSlot("init") == Object.getSlot("init");
};

// Unknown slot.
assert { b.locateSlot("ENOENT") == nil; };
b.getSlot("ENOENT");
[00041066:error] !!! lookup failed: ENOENT
```

- **hash**

A **Hash** object for **this**. This default implementation returns a different hash for every object, so every key maps to a different cells. Classes that have value semantic should override the hash method so as objects that are equal (in the **Object.'=='** sense) have the same hash. **String.hash** does so for instance; as a consequence different String objects with the same value map to the same cell.

A hash only makes sense as long as the hashed object exists.

urbiscript
Session

```
var o1 = Object.new();
var o2 = Object.new();
assert
{
  o1.hash == o1.hash;
  o1.hash != o2.hash;
};
```

- **hasLocalSlot(*slot*)**

Whether `this` features a slot *slot*, locally (not from some ancestor). See also [hasSlot](#).

```
class Base { var this.base = 23; } |;
class Derive: Base { var this.derive = 43 } |;
assert(Derive.hasLocalSlot("derive"));
assert(!Derive.hasLocalSlot("base"));
```

urbiscript
Session

- **hasProperty(*slotName*, *propName*)**

Whether the slot *slotName* of `this` has a property *propName*.

```
const var halfPi = pi / 2|;
assert
{
  hasProperty("halfPi", "constant");
  !hasProperty("halfPi", "foobar");
};
```

urbiscript
Session

- **hasSlot(*slot*)**

Whether `this` has the slot *slot*, locally, or from some ancestor. See also [hasLocalSlot](#).

```
Derive.hasSlot("derive");
Derive.hasSlot("base");
!Base.hasSlot("derive");
```

Assertion
Block

- **'\$id'**

- **inspect(*deep* = false)**

Describe `this`: its prototypes and slots, and their properties. If *deep*, all the slots are described, not only the local slots. See also [dump](#).

```
do (2) { var this.attr = "foo"; this.attr->prop = "bar" }.inspect;
[00001227] *** Inspecting 2
[00001227] *** ** Prototypes:
[00001227] ***   0
[00001227] *** ** Local Slots:
[00001228] ***   attr : String
[00001228] ***     Properties:
[00001228] ***       prop : String = "bar"
```

urbiscript
Session

- **isA(*obj*)**

Whether `this` has *obj* in his parents.

```
Float.isA(Orderable);
! String.isA(Float);
```

Assertion
Block

- **isNil**

Whether `this` is `nil`.

```
nil.isNil;
! 0.isNil;
```

Assertion
Block

- **isProto**

Whether `this` is a prototype.

```
Float.isProto;
! 42.isProto;
```

Assertion
Block

- **isVoid**

Whether `this` is `void`. See `void`.

Assertion Block

```
void.isVoid;
! 42.isVoid;
```

- **localSlotNames**

A list with the names of the local (i.e., not including those of its ancestors) slots of `this`. See also `slotNames`.

urbiscript Session

```
var top = Object.new|;
var top.top1 = 1|;
var top.top2 = 2|;
var bot = top.new|;
var bot.bot1 = 10|;
var bot.bot2 = 20|;
assert
{
    top.localSlotNames == ["top1", "top2"];
    bot.localSlotNames == ["bot1", "bot2"];
};
```

- **locateSlot(*slot*)**

The `Object` that provides *slot* to `this`, or `nil` if `this` does not feature *slot*.

Assertion Block

```
locateSlot("locateSlot") == Object;
locateSlot("doesNotExist").isNil;
```

- **print**

Send `this` to the `Channel.topLevel` channel.

urbiscript Session

```
1.print;
[00001228] 1
[1, "12"].print;
[00001228] [1, "12"]
```

- **protos**

The list of prototypes of `this`.

Assertion Block

```
12.protos == [Float];
```

- **properties(*slotName*)**

A dictionary of the properties of slot *slotName*. Raise an error if the slot does not exist.

urbiscript Session

```
2.properties("foo");
[00238495:error] !!! lookup failed: foo
do (2) { var foo = "foo" }.properties("foo");
[00238501] ["constant" => false]
do (2) { var foo = "foo" ; foo->bar = "bar" }.properties("foo");
[00238502] ["bar" => "bar", "constant" => false]
```

- **removeLocalSlot(*slot*)**

Remove *slot* from the (local) list of slots of `this`, and return `this`. Raise an error if *slot* does not exist. See also `removeSlot`.

urbiscript Session

```
var base = Object.new|;
var base.slot = "base"|;
var derive = Base.new|;
var derive.slot = "derive"|;
```

```

derive.removeLocalSlot("foo");
[00000080:error] !!! lookup failed: foo

assert
{
    derive.removeLocalSlot("slot") === derive;
    derive.localSlotNames == [];
    base.slot == "base";
};

derive.removeLocalSlot("slot");
[00000090:error] !!! lookup failed: slot

assert
{
    base.slot == "base";
};

```

- **removeProperty(slotName, propName)**

Remove the property *propName* from the slot *slotName*. Raise an error if the slot does not exist. Warn if *propName* does not exist; in a future release this will be an error.

```

var r = Object.new();
// Non-existing slot.
r.removeProperty("slot", "property");
[00000072:error] !!! lookup failed: slot

var r.slot = "slot value";
// Non-existing property.
r.removeProperty("slot", "property");
[00000081:warning] !!! no such property: slot->property
[00000081:warning] !!!      called from: removeProperty

r.slot->property = "property value";
assert
{
    r.hasProperty("slot", "property");
    // Existing property.
    r.removeProperty("slot", "property") == "property value";
    ! r.hasProperty("slot", "property");
};

```

urbiscript
Session

- **removeProto(proto)**

Remove *proto* from the list of prototypes of *this*, and return *this*. Do nothing if *proto* is not a prototype of *this*.

```

do (Object.new)
{
    assert
    {
        addProto(Orderable);
        removeProto(123) === this;
        protos == [Orderable, Object];
        removeProto(Orderable) === this;
        protos == [Object];
    };
}();

```

urbiscript
Session

- **removeSlot(slot)**

Remove *slot* from the (local) list of slots of *this*, and return *this*. Warn if *slot* does not exist; in a future release this will be an error. See also [removeLocalSlot](#).

urbiscript
Session

```
{
  var base = Object.new;
  var base.slot = "base";

  var derive = Base.new;
  var derive.slot = "derive";

  assert
  {
    derive.removeSlot("foo") === derive;
[00000080:warning] !!! no such local slot: foo
[00000080:warning] !!!      called from: removeSlot
[00000080:warning] !!!      called from: code
[00000080:warning] !!!      called from: eval
[00000080:warning] !!!      called from: value
[00000080:warning] !!!      called from: assertCall

    derive.removeSlot("slot") === derive;
    derive.localSlotNames == [];
    base.slot == "base";
    derive.removeSlot("slot") === derive;
[00000099:warning] !!! no such local slot: slot
[00000099:warning] !!!      called from: removeSlot
[00000099:warning] !!!      called from: code
[00000099:warning] !!!      called from: eval
[00000099:warning] !!!      called from: value
[00000099:warning] !!!      called from: assertCall

    base.slot == "base";
  };
};
}
```

- **setConstSlot**

Like `setSlot` but the created slot is const.

urbiscript
Session

```
assert(setConstSlot("fortyTwo", 42) == 42);
fortyTwo = 51;
[00000000:error] !!! cannot modify const slot
```

- **setProperty(*slotName*, *propName*, *value*)**

Set the property *propName* of slot *slotName* to *value*. Raise an error in *slotName* does not exist. Return *value*. This is what *slotName*->*propName* = *value* actually performs.

urbiscript
Session

```
do (Object.new)
{
  var slot = "slot";
  var value = "value";
  assert
  {
    setProperty("slot", "prop", value) === value;
    "prop" in properties("slot");
    getProperty("slot", "prop") === value;
    slot->prop === value;
    setProperty("slot", "noSuchProperty", value) === value;
  };
};

setProperty("noSuchSlot", "prop", "12");
[00000081:error] !!! lookup failed: noSuchSlot
```

- **setProtos(*protos*)**

Set the list of prototypes of `this` to *protos*. Return `void`.

urbiscript
Session

```
do (Object.new)
{
  assert
  {
    protos == [Object];
    setProtos([Orderable, Object]).isVoid;
    protos == [Orderable, Object];
  };
}!;
```

- **setSlot(*name*, *value*)**

Create a slot *name* mapping to *value*. Raise an error if *name* was already defined. This is what `var name = value` actually performs.

```
Object.setSlot("theObject", Object) === Object;
Object.theObject === Object;
theObject === Object;
```

Assertion Block

If the current job is in redefinition mode, `setSlot` on an already defined slot is not an error and overwrites the slot like `updateSlot` would. See the `redefinitionMode` method in [System](#).

- **slotNames**

A list with the slot names of `this` and its ancestors.

```
Object.localSlotNames
  .subset(Object.slotNames);
Object.protos.foldl(function (var r, var p) { r + p.localSlotNames },
  [])
  .subset(Object.slotNames);
```

Assertion Block

- **type**

The name of the type of `this`. The `class` construct defines this slot to the name of the class ([Section 12.4](#)). This is used to display the name of “instances”.

```
class Example {};
[00000081] Example
assert
{
  Example.type == "Example";
};
Example.new;
[00000081] Example_0x6fb2720
```

urbiscript Session

- **uid**

The unique id of `this`.

```
{
  var foo = Object.new;
  var bar = Object.new;
  assert
  {
    foo.uid == foo.uid;
    foo.uid != bar.uid;
  };
};
```

urbiscript Session

- **unacceptVoid**

Return `this`. See `void` to know why.

urbiscript Session

```
{
    var o = Object.new|
    assert(o.unacceptVoid === o);
};
```

- `updateSlot(name, value)`

Map the existing slot named *name* to *value*. Raise an error if *name* was not defined.

Assertion Block

```
Object.setSlot("one", 1) == 1;
Object.updateSlot("one", 2) == 2;
Object.one == 2;
```

- `'!'`

Logical negation. If `this` evaluates to false return `true` and vice-versa.

Assertion Block

```
!1 == false;
!0 == true;

!"foo" == false;
!"'" == true;
```

- `'+='(that)`

Bounce to `this '+' that`.

- `'-='(that)`

Bounce to `this '-' that`.

- `'*='(that)`

Bounce to `this '*' that`.

- `'/='(that)`

Bounce to `this '/' that`.

- `'^='(that)`

Bounce to `this '^' that`.

- `'%='(that)`

Bounce to `this '%-' that`.

- `'==='(that)`

Whether `this` and `that` are equal. See also [Comparable](#) and [Section 23.1.8.6](#). By default, bounces to `'==='`. This operator *must* be redefined for objects that have a value-semantics; for instance two `String` objects that denotes the same string should be equal according to `==`, although physically different (i.e., not equal according to `===`).

urbiscript Session

```
{
    var o1 = Object.new;
    var o2 = Object.new;
    assert
    {
        o1 == o1;
        !(o1 == o2);
        o1 != o2;
        !(o1 != o1);

        1 == 1;
        "1" == "1";
        [1] == [1];
    };
};
```

- `'=='`(*that*)

Whether `this` and `that` are exactly the same object (i.e., `this` and `that` are two different means to denote the very same location in memory). To denote equivalence, use `'=='`; for instance two `Float` objects that denote 42 can be different objects (in the sense of `==`), but will be considered equal by `==`. See also `'=='`, and [Section 23.1.8.6](#).

```
{
  var o1 = Object.new;
  var o2 = Object.new;
  assert
  {
    o1 === o1;
    !(o1 === o2);

    !(1 === 1);
    !(1" === "1");
    !([1] === [1]);
  };
}
```

urbiscript
Session

- `'!='`(*that*)

The negation of `\this == \that`, see `'=='`.

```
{
  var o1 = Object.new;
  var o2 = Object.new;
  assert
  {
    o1 !== o2;
    !(o1 !== o1);

    1 !== 1;
    1" !== "1";
    [1] !== [1];
  };
}
```

urbiscript
Session

24.41 Orderable

Objects that have a concept of “less than”. See also [Comparable](#).

This object, made to serve as prototype, provides a definition of `<` based on `>`, and vice versa; and definition of `<=/>=` based on `</>==`. You **must** define either `<` or `>`, otherwise invoking either method will result in endless recursions.

```
class Foo : Orderable
{
  var value = 0;
  function init (v) { value = v; };
  function '<' (that) { value < that.value; };
  function asString() { "<" + value.asString + ">"; };
};

var one = Foo.new(1);
var two = Foo.new(2);

assert
{
  one <= one ; one <= two ; !(two <= one);
  !(one > one) ; !(one > two) ; two > one;
  (one >= one) ; !(one >= two) ; two >= one;
};
```

urbiscript
Session

24.42 OutputStream

OutputStreams are used to write (possibly binary) files by hand.

24.42.1 Prototypes

- [Stream](#)

24.42.2 Construction

An OutputStream is a writing-interface to a file; its constructor requires a [File](#). If the file already exists, content is *appended* to it. Remove the file beforehand if you want to override its content.

urbiscript
Session

```
var o1 = OutputStream.new(File.create("file.txt"));
[00000001] OutputStream_Ox827000

var o2 = OutputStream.new(File.new("file.txt"));
[00000002] OutputStream_Ox827000
```

When a stream ([OutputStream](#) or [InputStream](#)) is opened on a File, that File cannot be removed. On Unix systems, this is handled gracefully (the references to the file are removed, but the content is still there for the streams that were already bound to this file); so in practice, the File appears to be removable. On Windows, the File cannot be removed at all. Therefore, do not forget to close the streams you opened.

urbiscript
Session

```
o1.close;
o2.close;
```

24.42.3 Slots

- Output `this.asString`. Return `this` to enable chains of calls. Raise an error if the file is closed.

urbiscript
Session

```
var o = OutputStream.new(File.create("fresh.txt"))|;
o << 1 << "2" << [3, [4]]|;
o.close;
assert (File.new("fresh.txt").content.data == "12[3, [4]]");
o << 1;
[00000005:error] !!! <<: stream is closed
```

- `flush`

To provide efficient input/output operations, *buffers* are used. As a consequence, what is put into a stream might not be immediately saved on the actual file. To *flush* a buffer means to dump its content to the file. Raise an error if the file is closed.

urbiscript
Session

```
var s = OutputStream.new(File.create("file.txt"))|
s.flush;
s.close;
s.flush;
[00039175:error] !!! flush: stream is closed
```

- `put(byte)`

Output the character corresponding to the numeric code `byte` in `this`, and return `this`. Raise an error if the file is closed.

urbiscript
Session

```
var f = File.create("put.txt") |
var os = OutputStream.new(f) |
assert
```

```
{
    os.put(0)
        .put(255)
        .put(72).put(101).put(108).put(108).put(111)
    === os;
    f.content.data == "\0\xffHello";
};

os.put(12.5);
[00029816:error] !!! put: argument 1: bad numeric conversion: overflow or non empty fractional part: 12
os.put(-1);
[00034840:error] !!! put: argument 1: bad numeric conversion: negative overflow: -1
os.put(256);
[00039175:error] !!! put: argument 1: bad numeric conversion: positive overflow: 256
os.close;
os.put(0);
[00039179:error] !!! put: stream is closed
```

24.43 Pair

A *pair* is a container storing two objects, similar in spirit to `std::pair` in C++.

24.43.1 Prototype

- [Tuple](#)

24.43.2 Construction

A *Pair* is constructed with two arguments.

```
Pair.new(1, 2);
[00000001] (1, 2)

Pair.new;
[00000003:error] !!! new: expected 2 arguments, given 0

Pair.new(1, 2, 3, 4);
[00000003:error] !!! new: expected 2 arguments, given 4
```

urbscript
Session

24.43.3 Slots

- `first`

Return the first member of the pair.

```
Pair.new(1, 2).first == 1;
Pair[0] === Pair.first;
```

Assertion
Block

- `second`

Return the second member of the pair.

```
Pair.new(1, 2).second == 2;
Pair[1] === Pair.second;
```

Assertion
Block

24.44 Path

A *Path* points to a file system entity (directory, file and so forth).

24.44.1 Prototypes

- Comparable
- Orderable

24.44.2 Construction

A Path is constructed with the string that points to the file system entity. This path can be relative or absolute.

urbiscript
Session

```
Path.new("/path/file.u");
[00000001] Path("/path/file.u")
```

Some minor simplifications are made, such as stripping useless ‘./’ occurrences.

urbiscript
Session

```
Path.new("./../../../../foo/");
[00000002] Path("foo")
```

24.44.3 Slots

- absolute

Whether `this` is absolute.

Assertion
Block

```
Path.new("/abs/path").absolute;
!Path.new("rel/path").absolute;
```

- asList

List of names used in path (directories and possibly file), from bottom up. There is no difference between relative path and absolute path.

Assertion
Block

```
Path.new("/path/to/file.u").asList == ["path", "to", "file.u"];
Path.new("/path").asList           == Path.new("path").asList;
```

- asPrintable

Assertion
Block

```
Path.new("file.txt").asPrintable == "Path(\"file.txt\")";
```

- asString

The name of the file.

Assertion
Block

```
Path.new("file.txt").asString == "file.txt";
```

- basename

Base name of the path.

Assertion
Block

```
Path.new("/absolute/path/file.u").basename == "file.u";
Path.new("relative/path/file.u").basename == "file.u";
```

- cd

Change current working directory to `this`. Return the new current working directory as a Path.

- cwd

The current working directory.

urbiscript
Session

```
{
    // Save current directory.
    var pwd = Path.cwd();
    // Go into '/'.
    var root = Path.new("/").cd();
    // Current working directory is '/'.
    assert(Path.cwd == root);
    // Go back to the directory we were in.
    assert(pwd.cd == pwd);
};
```

- **dirname**

Directory name of the path.

```
Path.new("/abs/path/file.u").dirname == Path.new("/abs/path");
Path.new("rel/path/file.u").dirname == Path.new("rel/path");
```

Assertion Block

- **exists**

Whether something (a file, a directory, ...) exists where `this` points to.

```
Path.cwd.exists;
Path.new("/").exists;
!Path.new("/this/path/does/not/exists").exists;
```

Assertion Block

- **isDir**

Whether `this` is a directory.

```
Path.cwd.isDir;
```

Assertion Block

- **isReg**

Whether `this` is a regular file.

```
!Path.cwd.isReg;
```

Assertion Block

- **open**

Open `this`. Return either a *Directory* or a *File* according the type of `this`. See [File](#) and [Directory](#).

- **readable**

Whether `this` is readable. Throw if does not even exist.

```
Path.new(".").readable;
```

Assertion Block

- **writable**

Whether `this` is writable. Throw if does not even exist.

```
Path.new(".").writable;
```

Assertion Block

- **'/' (*rhs*)**

Create a new *Path* that is the concatenation of `this` and *rhs*. *rhs* can be a *Path* or a *String* and cannot be absolute.

```
assert(Path.new("/foo/bar") / Path.new("baz/qux/quux")
      == Path.new("/foo/bar/baz/qux/quux"));
Path.cwd / Path.new("/tmp/foo");
[00000003:error] !!! /: Rhs of concatenation is absolute: /tmp/foo
```

urbiscript Session

- `'==' (that)`

Same as comparing the string versions of `this` and `that`. Beware that two paths may be different and point to the very same location.

Assertion Block

```
Path.new("/a") == Path.new("/a");
!(Path.new("/a") == Path.new("a"));
```

- `'<' (that)`

Same as comparing the string versions of `this` and `that`.

Assertion Block

```
Path.new("/a") < Path.new("/a/b");
!(Path.new("/a/b") < Path.new("/a"));
```

24.45 Pattern

`Pattern` class is used to make correspondences between a pattern and another `Object`. The visit is done either on the pattern or on the element against which the pattern is compared.

`Patterns` are used for the implementation of the pattern matching. So any class made compatible with the pattern matching implemented by this class will allow you to use it implicitly in your scripts.

urbiscript Session

```
[1, var a, var b] = [1, 2, 3];
[00000000] [1, 2, 3]
a;
[00000000] 2
b;
[00000000] 3
```

24.45.1 Prototypes

- `Object`

24.45.2 Construction

A `Pattern` can be created with any object that can be matched.

urbiscript Session

```
Pattern.new([1]); // create a pattern to match the list [1].
[00000000] Pattern_0x189ea80
Pattern.new(Pattern.Binding.new("a")); // match anything into "a".
[00000000] Pattern_0x18d98b0
```

24.45.3 Slots

- `Binding`

A class used to create pattern variables.

urbiscript Session

```
Pattern.Binding.new("a");
[00000000] var a
```

- `bindings`

A `Dictionary` filled by the match function for each `Binding` contained inside the pattern.

urbiscript Session

```
{
  var p = Pattern.new([Pattern.Binding.new("a"), Pattern.Binding.new("b")]);
  assert (p.match([1, 2]));
  p.bindings
};
```

[00000000] ["a" => 1, "b" => 2]

- `match(value)`

Use `value` to unify the current pattern with this value. Return the status of the match.

- If the match is correct, then the `bindings` member will contain the result of every matched values.
- If the match is incorrect, then the `bindings` member should not be used.

If the pattern contains multiple `Binding` with the same name, then the behavior is undefined.

```
Pattern.new(1).match(1);
Pattern.new([1, 2]).match([1, 2]);
! Pattern.new([1, 2]).match([1, 3]);
! Pattern.new([1, 2]).match([1, 2, 3]);
Pattern.new(Pattern.Binding.new("a")).match(0);
Pattern.new([1, Pattern.Binding.new("a")]).match([1, 2]);
! Pattern.new([1, Pattern.Binding.new("a")]).match(0);
```

Assertion Block

- `matchPattern(pattern, value)`

This function is used as a callback function to store all bindings in the same place. This function is useful inside objects that implement a `match` or `matchAgainst` function that need to continue the match deeper. Return the status of the match (a Boolean).

The `pattern` should provide a method `match(handler, value)` otherwise the value method `matchAgainst(handler, pattern)` is used. If none are provided the `'=='` operator is used.

To see how to use it, you can have a look at the implementation of `List.matchAgainst`.

- `pattern`

The pattern given at the creation.

```
Pattern.new(1).pattern == 1;
Pattern.new([1, 2]).pattern == [1, 2];
{
    var pattern = [1, Pattern.Binding.new("a")];
    Pattern.new(pattern).pattern === pattern
};
```

Assertion Block

24.46 Position

This class is used to handle file locations with a line, column and file name.

24.46.1 Prototypes

- `Object`

24.46.2 Construction

Without argument, a newly constructed Position has its fields initialized to the first line and the first column.

```
Position.new;
[00000001] 1.1
```

urbiscript Session

With a position argument `p`, the newly constructed Position is a clone of `p`.

urbiscript Session

```
Position.new(Position.new(2, 3));
[00000001] 2.3
```

With two float arguments l and c , the newly constructed Position has its line and column defined and an empty file name.

urbiscript
Session

```
Position.new(2, 3);
[00000001] 2.3
```

With three arguments f , l and c , the newly constructed Position has its file name, line and column defined.

urbiscript
Session

```
Position.new("file.u", 2, 3);
[00000001] file.u:2.3
```

24.46.3 Slots

- `'+'(n)`

Return a new Position which is shifted from n columns to the right. The minimal value of the new position column is 1.

Assertion
Block

```
Position.new(2, 3) + 2 == Position.new(2, 5);
Position.new(2, 3) + -4 == Position.new(2, 1);
```

- `'-'(n)`

Return a new Position which is shifted from n columns to the left. The minimal value of the new Position column is 1.

Assertion
Block

```
Position.new(2, 3) - 1 == Position.new(2, 2);
Position.new(2, 3) - -4 == Position.new(2, 7);
```

- `'=='(other)`

Compare the lines and columns of two Positions.

Assertion
Block

```
Position.new(2, 3) == Position.new(2, 3);
Position.new("a.u", 2, 3) == Position.new("b.u", 2, 3);
Position.new(2, 3) != Position.new(2, 2);
```

- `'<'(other)`

Order comparison of lines and columns.

Assertion
Block

```
Position.new(2, 3) < Position.new(2, 4);
Position.new(2, 3) < Position.new(3, 1);
```

- `asString`

Present as `'file:line.column'`, the file name is omitted if it is not defined.

Assertion
Block

```
Position.new("file.u", 2, 3).asString == "file.u:2.3";
```

- `column`

Field which give access to the column number of the Position.

Assertion
Block

```
Position.new(2, 3).column == 3;
```

- `columns(n)`

Identical to `'+'(n)`.

Assertion
Block

```
Position.new(2, 3).columns(2) == Position.new(2, 5);
Position.new(2, 3).columns(-4) == Position.new(2, 1);
```

- **file**

The Path of the Position file.

```
Position.new("file.u", 2, 3).file == Path.new("file.u");
Position.new(2, 3).file == nil;
```

Assertion
Block

- **line**

Field which give access to the line number of the Position.

```
Position.new(2, 3).line == 2;
```

Assertion
Block

- **lines(*n*)**

Add *n* lines and reset the column number to 1.

```
Position.new(2, 3).lines(2) == Position.new(4, 1);
Position.new(2, 3).lines(-1) == Position.new(1, 1);
```

Assertion
Block

24.47 Primitive

C++ routine callable from urbiscript.

24.47.1 Prototypes

- [Executable](#)

24.47.2 Construction

It is not possible to construct a Primitive.

24.47.3 Slots

- [apply\(*args*\)](#)

Invoke a primitive. The argument list, *args*, must start with the target.

```
Float.getSlot("+").isA(Global.getSlot("Primitive"));
Float.getSlot("+").apply([1, 2]) == 3;

String.getSlot("+").isA(Global.getSlot("Primitive"));
String.getSlot("+").apply(["1", "2"]);
```

Assertion
Block

- [asPrimitive](#)

Return `this`.

```
Float.getSlot("+").asPrimitive === Float.getSlot("+");
```

Assertion
Block

24.48 Process

A Process is a separated task handled by the underneath operating system.

Windows Issues

Process is not yet supported under Windows.

24.48.1 Prototypes

- [Object](#)

24.48.2 Example

The following examples runs the `cat` program, a Unix standard command that simply copies on its (standard) output its (standard) input.

```
urbiscript
Session
var p = Process.new("cat", []);
[00000004] Process cat
```

Just created, this process is not running yet. Use `run` to launch it.

```
urbiscript
Session
p.status;
[00000005] not started

p.run;
p.status;
[00000006] running
```

Then we feed its input, named `stdin` in the Unix tradition, and close its input.

```
urbiscript
Session
p.stdin << "1\n" |
p.stdin << "2\n" |
p.stdin << "3\n" |;

p.status;
[00000007] running

p.stdin.close;
```

At this stage, the status of the process is unknown, as it is running asynchronously. If it has had enough time to “see” that its input is closed, then it will have finished, otherwise we might have to wait for awhile. The method `join` means “wait for the process to finish”.

```
urbiscript
Session
p.join;

p.status;
[00000008] exited with status 0
```

Finally we can check its output.

```
urbiscript
Session
p.stdout.asList;
[00000009] ["1", "2", "3"]
```

24.48.3 Construction

A Process needs a program name to run and a possibly-empty list of command line arguments. Calling `run` is required to execute the process.

```
urbiscript
Session
Process.new("cat", []);
[00000004] Process cat

Process.new("cat", ["--version"]);
[00000004] Process cat
```

24.48.4 Slots

- `asProcess`

Return `this`.

```
urbiscript
Session
do (Process.new("cat", []))
{
    assert (asProcess === this);
}|;
```

- **asString**

Process and the name of the program.

```
Process.new("cat", ["--version"]).asString
  == "Process cat";
```

Assertion Block

- **done**

Whether the process has completed its execution.

```
do (Process.new("sleep", ["1"]))
{
    assert (!done);
    run;
    assert (!done);
    join;
    assert (done);
}|;
```

urbiscript Session

- **join**

Wait for the process to finish. Changes its status.

```
do (Process.new("sleep", ["2"]))
{
    var t0 = System.time;
    assert (status.asString == "not started");
    run;
    assert (status.asString == "running");
    join;
    assert (t0 + 2s <= System.time);
    assert (status.asString == "exited with status 0");
}|;
```

urbiscript Session

- **kill**

If the process is not `done`, interrupt it (with a SIGKILL in Unix parlance). You still have to wait for its termination with `join`.

```
do (Process.new("sleep", ["1"]))
{
    run;
    kill;
    join;
    assert (done);
    assert (status.asString == "killed by signal 9");
}|;
```

urbiscript Session

- **name**

The (base) name of the program the process runs.

```
Process.new("cat", ["--version"]).name == "cat";
```

Assertion Block

- **run**

Launch the process. Changes its status. A process can only be run once.

```
do (Process.new("sleep", ["1"]))
{
    assert (status.asString == "not started");
    run;
    assert (status.asString == "running");
    join;
    assert (status.asString == "exited with status 0");
    run;
}|;
[00021972:error] !!! run: Process was already run
```

urbiscript Session

- `runTo`

- `status`

An object whose slots describe the status of the process.

- `stderr`

An `InputStream` (the output of the Process is an input for Urbi) to the standard error stream of the process.

urbiscript
Session

```
do (Process.new("urbi-send", ["--no-such-option"]))
{
    run;
    join;
    assert
    {
        stderr.asList ==
        ["urbi-send: invalid option: --no-such-option",
         "Try 'urbi-send --help' for more information."];
    };
}|;
```

- `stdin`

An `OutputStream` (the input of the Process is an output for Urbi) to the standard input stream of the process.

urbiscript
Session

```
do (Process.new(System.programName, ["--version"]))
{
    run;
    join;
    assert
    {
        stdout.asList[1] == "Copyright (C) 2004-2012 Gostai S.A.S..";
    };
}|;
```

- `stdout`

An `InputStream` (the output of the Process is an input for Urbi) to the standard output stream of the process.

urbiscript
Session

```
do (Process.new("cat", []))
{
    run;
    stdin << "Hello, World!\n";
    stdin.close;
    join;
    assert (stdout.asList == ["Hello, World!"]);
}|;
```

24.49 Profile

A `Profile` object contains information about the efficiency of a piece of code.

24.49.1 Example

24.49.1.1 Basic profiling

One can profile a piece of code with the `System.profile` function.

urbiscript
Session

```

var profile = System.profile(function() { echo("foo") });
[00000001] *** foo
[00001672] Profile(
  Yields:          0
  Total time (us):    112
  Wall-clock time (us): 112
  Function calls:     16
  Max depth:         5

  -----
  |   function | % | cumulative | self | calls | self | 
  |           |   | (us)       | (us) |       | (us/call) |
  |-----+-----+-----+-----+-----+-----+
  |   send   | 24.11 |      27 |    27 |    1 |    27 |
  |   echo   | 14.29 |      43 |    16 |    1 |    16 |
  |   apply  | 10.71 |      55 |    12 |    1 |    12 |
  |   apply  |  9.82 |      66 |    11 |    1 |    11 |
  |   +     |  8.93 |      76 |    10 |    2 |     5 |
  | <profiled> | 6.25 |      83 |     7 |    1 |     7 |
  |   asString | 6.25 |      90 |     7 |    1 |     7 |
  |   lobby   | 5.36 |      96 |     6 |    2 |     3 |
  |   getSlot | 3.57 |     104 |     4 |    1 |     4 |
  |   +     | 2.68 |     107 |     3 |    1 |     3 |
  |   lobby   | 2.68 |     110 |     3 |    2 |     1 |
  |   Lobby   | 1.79 |     112 |     2 |    2 |     1 |
  |-----+-----+-----+-----+-----+-----+
)
)

```

The result is a [Profile](#) object that contains information about which functions were used when evaluating the given code, how many times they were called, how much time was spent in them, ... Lines are sorted by decreasing “self time”. Note that the `<profiled>` special function stands for the function given in parameter. Every line is represented by a [Profile.Function](#) object, see its documentation for the meaning of every column.

24.49.1.2 Asynchronous profiling

If the profiled code spawns asynchronous tasks via `detach` or `at` for instance, additional statistics will be included in the resulting [Profile](#) every time the detached code is executed. This is extremely useful to profile asynchronous code based on `at` for instance.

```

var Global.x = false;
function profiled()
{
  at (x)
    echo("true")
  onleave
    echo("false")
};

// This is the profiling for the creation of the 'at'. Note that the
// condition was already evaluated once, to see whether it should trigger
// immediately.
var profile_async = System.profile(&profiled);
[00000000] Profile(
  Yields:          0
  Total time (us):    485
  Wall-clock time (us): 485
  Function calls:     15
  Max depth:         7

  -----
  |   function | % | cumulative | self | calls | self | 
  |           |   | (us)       | (us) |       | (us/call) |
  |-----+-----+-----+-----+-----+-----+
)
)

```

urbiscript
Session

```

|-----+-----+-----+-----+-----+-----+
| <profiled> | 38.14 |    185 |    185 |      1 |    185 | |
|   new | 13.20 |    249 |     64 |      1 |     64 |
|   at: { x } | 12.37 |    309 |     60 |      1 |     60 |
|   onEvent | 10.10 |    358 |     49 |      1 |     49 |
| callMessage | 5.77 |    386 |     28 |      1 |     28 |
|   map | 5.15 |    411 |     25 |      1 |     25 |
| evalArgs | 3.92 |    430 |     19 |      1 |     19 |
|   clone | 3.30 |    446 |     16 |      1 |     16 |
| updateSlot | 2.47 |    458 |     12 |      1 |     12 |
|   each| | 2.06 |    468 |     10 |      1 |     10 |
| getSlot | 1.44 |    475 |      7 |      1 |      7 |
|   init | 0.82 |    479 |      4 |      1 |      4 |
|   x | 0.82 |    483 |      4 |      2 |      2 |
| args | 0.41 |    485 |      2 |      1 |      2 |
|-----+-----+-----+-----+-----+-----+
)

// Trigger the at twice.
Global.x = true;
[00106213] true
[00106213] *** true
Global.x = false;
[00172119] false
[00172119] *** false

// The profile now includes additional statistic about the evaluations of
// the condition and the bodies of the at.
profile_async;
[00178623] Profile(
  Yields:                      2
  Total time (us):           1307
  Wall-clock time (us):       1307
  Function calls:             51
  Max depth:                  7

  .
  .
  .
  |-----+-----+-----+-----+-----+-----+
  | function | % | cumulative | self | calls | self | (us/call) |
  |          |   | (us)       | (us) |       |       |           |
  |-----+-----+-----+-----+-----+-----+
  |   event | 15.61 |    204 |    204 |      1 |    204 | |
  |   event | 15.00 |    400 |    196 |      1 |    196 |
  | <profiled> | 14.15 |    585 |    185 |      1 |    185 |
  |   at: { x } | 9.26 |    706 |    121 |      3 | 40.333 |
  |   echo | 6.12 |    786 |     80 |      2 |     40 |
  |   apply | 5.66 |    860 |     74 |      2 |     37 |
  |   lobby | 4.97 |    925 |     65 |      4 | 16.250 |
  |   new | 4.90 |    989 |     64 |      1 |     64 |
  |   onEvent | 3.75 |   1038 |     49 |      1 |     49 |
  |   send | 2.60 |   1072 |     34 |      2 |     17 |
  | callMessage | 2.14 |   1100 |     28 |      1 |     28 |
  | getSlot | 1.99 |   1126 |     26 |      3 | 8.667 |
  |   map | 1.91 |   1151 |     25 |      1 |     25 |
  |   + | 1.53 |   1171 |     20 |      4 |      5 |
  | evalArgs | 1.45 |   1190 |     19 |      1 |     19 |
  |   + | 1.30 |   1207 |     17 |      2 | 8.500 |
  |   clone | 1.22 |   1223 |     16 |      1 |     16 |
  | asString | 1.07 |   1237 |     14 |      2 |      7 |
  |   apply | 1.07 |   1251 |     14 |      2 |      7 |
  | updateSlot | 0.92 |   1263 |     12 |      1 |     12 |
  |   lobby | 0.84 |   1274 |     11 |      4 | 2.750 |
  |   each| | 0.77 |   1284 |     10 |      1 |     10 |
  | Lobby | 0.54 |   1291 |      7 |      4 | 1.750 |
  |   x | 0.46 |   1297 |      6 |      3 |      2 |
  | init | 0.31 |   1301 |      4 |      1 |      4 |
  |-----+-----+-----+-----+-----+-----+
)

```

```
|      x | 0.31 | 1305 | 4 | 1 | 4 |
| args | 0.15 | 1307 | 2 | 1 | 2 |
,-----,-----,-----,-----,-----,-----,
)
```

24.49.2 Prototypes

- [Object](#)

24.49.3 Construction

Profile objects are not meant to be cloned as they are created by [System.profile](#) internal machinery.

24.49.4 Slots

- [calls](#)

Return a [List](#) of [Profile.Function](#) objects. Each element of this list describes, for a given function, statistics about how many times it is called and how much time is spent in it.

- [maxFunctionCallDepth](#)

The maximum function call depth reached.

```
// Example continued from Construction.
profile.maxFunctionCallDepth == 5;
```

Assertion Block

- [Function](#)

See [Profile.Function](#).

- [totalCalls](#)

The total number of function calls made.

```
// Example continued from Construction.
profile.totalCalls == 16;
```

Assertion Block

- [totalTime](#)

The total CPU time. It can be higher than the wall clock time on multi-core processors for instance.

- [wallClockTime](#)

The time spent between the beginning and the end as if measured on a wall clock.

- [yields](#)

The scheduler has to execute many coroutines in parallel. A coroutine yields when it gives the opportunity to another to be executed until this one yields and so on... This slot contains the number of scheduler yields.

```
// Example continued from Construction.
profile.yields == 0;
```

Assertion Block

24.50 Profile.Function

A Function object contains information about calls of a given function during a profiling operation.

24.50.1 Prototypes

- [Object](#)

24.50.2 Construction

Function objects are not meant to be cloned as they are created by [System.profile](#) internal machinery.

urbiscript
Session

```
function Float.fact()
{
    if (this <= 1)
        this
    else
        this * (this - 1).fact;
}|;
var profile = System.profile(function() { 20.fact });
[00009050] Profile(
    Yields:          0
    Total time (us): 171
    Wall-clock time (us): 171
    Function calls: 79
    Max depth: 22

    .
    .
    .
    |   function   |   %   | cumulative | self   | calls   | self   |
    |           |       | (us)       | (us)   |         | (us/call) |
    |-----+-----+-----+-----+-----+-----+
    |   fact   | 70.18 |      120 |     120 |      20 |       6 |
    |   -      | 10.53 |      138 |      18 |      19 | 0.947 |
    |   <=    |  8.77 |      153 |      15 |      20 | 0.750 |
    |   *     |  8.19 |      167 |      14 |      19 | 0.737 |
    | <profiled> | 2.34 |      171 |      4 |      1 |       4 |
    ,-----,-----,-----,-----,-----,-----,
)
profile.calls[0];
[00123833] Function('fact', 20, 0.000120, 0.000006)
```

24.50.3 Slots

- [calls](#)

The number of times this function was called during the profiling.

Assertion
Block

```
// Example continued from Construction section.
profile.calls[0].calls == 20;
```

- [name](#)

The name of the function called.

Assertion
Block

```
// Example continued from Construction section.
profile.calls[0].name == "fact";
```

- [selfTimePer](#)

Average CPU time spent in one function call. It is computed as the ratio of [selfTime](#) divided by [calls](#).

Assertion
Block

```
// Example continued from Construction section.
do (profile.calls[0])
{
    selfTimePer == selfTime / calls;
}
```

- **selfTime**

Total CPU time spent in all calls of the function.

```
// Example continued from Construction section.
profile.calls[0].selfTime.isA(Float);
```

Assertion Block

24.51 PseudoLazy

24.52 PubSub

PubSub provides an abstraction over Barrier **Barrier** to queue signals for each subscriber.

24.52.1 Prototypes

- **Object**

24.52.2 Construction

A PubSub can be created with no arguments. Values can be published and read by each subscriber.

```
var ps = PubSub.new;
[00000000] PubSub_0x28c1bc0
```

urbiscript Session

24.52.3 Slots

- **publish(ev)**

Queue the value *ev* to the queue of each subscriber. This method returns the value *ev*.

```
{
  var sub = ps.subscribe;
  assert
  {
    ps.publish(2) == 2;
    sub.getOne == 2;
  };
  ps.unsubscribe(sub)
};
```

urbiscript Session

- **subscribe**

Create a **Subscriber** and insert it inside the list of subscribers.

```
var sub = ps.subscribe |
ps.subscribers == [sub];
[00000000] true
```

urbiscript Session

- **Subscriber**

See [PubSub.Subscriber](#).

- **subscribers**

Field containing the list of **Subscriber** which are watching published values. This field only exists in instances of PubSub.

- **unsubscribe(sub)**

Remove a subscriber from the list of subscriber watching the published values.

```
ps.unsubscribe(sub) |
ps.subscribers;
[00000000] []
```

urbiscript Session

24.53 PubSub.Subscriber

`Subscriber` is created by `PubSub.subscribe`. It provides methods to access to the list of values published by `PubSub` instances.

24.53.1 Prototypes

- `Object`

24.53.2 Construction

A `PubSub.Subscriber` can be created with a call to `PubSub.subscribe`. This way of creating a `Subscriber` adds the subscriber as a watcher of values published on the instance of `PubSub`.

urbiscript
Session

```
var ps = PubSub.new |;
var sub = ps.subscribe;
[00000000] Subscriber_0x28607c0
```

24.53.3 Slots

- `getOne`

Block until a value is accessible and return it. If a value is already queued, then the method returns it without blocking.

urbiscript
Session

```
echo(sub.getOne) &
ps.publish(3);
[00000000] *** 3
```

- `getAll`

Block until a value is accessible. Return the list of queued values. If the values are already queued, then return them without blocking.

urbiscript
Session

```
ps.publish(4) |
ps.publish(5) |
echo(sub.getAll);
[00000000] *** [4, 5]
```

24.54 RangeIterable

This object is meant to be used as a prototype for objects that support an `asList` method, to use range-based `for` loops (Section 23.7.5.2).

24.54.1 Prototypes

- `Object`

24.54.2 Slots

- `all(fun)`

Return whether all the members of the target verify the predicate *fun*.

Assertion
Block

```
// Are all elements positive?
! [-2, 0, 2, 4].all(function (e) { e > 0 });
// Are all elements even?
[-2, 0, 2, 4].all(function (e) { e % 2 == 0 });
```

- `any(fun)`

Whether at least one of the members of the target verifies the predicate *fun*.

```
// Is there any even element?
! [-3, 1, -1].any(function (e) { e % 2 == 0 });
// Is there any positive element?
[-3, 1, -1].any(function (e) { e > 0 });
```

Assertion Block

- `each(fun)`

Apply the given functional value *fun* on all “members”, sequentially. Corresponds to range-`for` loops.

```
class range : RangeIterable
{
    var asList = [10, 20, 30];
}|;
for (var i : range)
    echo (i);
[00000000] *** 10
[00000000] *** 20
[00000000] *** 30
```

urbiscript Session

- `'each&'(fun)`

Apply the given functional value *fun* on all “members”, in parallel, starting all the computations simultaneously. Corresponds to range-`for&` loops.

```
{
    var res = [];
    for& (var i : range)
        res << i;
    assert(res.sort == [10, 20, 30]);
};
```

urbiscript Session

- `'each|'(fun)` Apply the given functional value *fun* on all “members”, with tight sequentially. Corresponds to range-`for|` loops.

```
{
    var res = [];
    for| (var i : range)
        res << i;
    assert(res == [10, 20, 30]);
};
```

urbiscript Session

24.55 Regexp

A `Regexp` is an object which allow you to match strings with a regular expression.

24.55.1 Prototypes

- `Container`
- `Object`

24.55.2 Construction

A `Regexp` is created with the regular expression once and for all, and it can be used many times to match with other strings.

```
Regexp.new(".");
[00000001] Regexp(".")
```

urbiscript Session

urbiscript supports Perl regular expressions, see [the perlre man page](#). Expressions cannot be empty.

urbiscript
Session

```
Regexp.new("");
[00000001:error] !!! new: invalid regular expression '' : Empty expression
```

24.55.3 Slots

- **asPrintable**

A string that shows that `this` is a Regexp, and its value.

Assertion
Block

```
Regexp.new("abc").asPrintable == "Regexp(\"abc\")";
Regexp.new("\d+(\.\d+)?").asPrintable == "Regexp(\"\\d+(\\.\\d+)?\")";
```

- **asString**

The regular expression that was compiled.

Assertion
Block

```
Regexp.new("abc").asString == "abc";
Regexp.new("\d+(\.\d+)?").asString == "\d+(\.\d+)?";
```

- **has(str)**

An experimental alias to `match`, so that the infix operators `in` and `not in` can be used (see [Section 23.1.8.7](#)).

Assertion
Block

```
"23.03"    in Regexp.new("^\\d+\\.\\d+$");
"-3.14" not in Regexp.new("^\\d+\\.\\d+$");
```

- **match(str)**

Whether `this` matches `str`.

urbiscript
Session

```
// Ordinary characters
var r = Regexp.new("oo")|
assert
{
  r.match("oo");
  r.match("foobar");
  !r.match("bazquux");
};

// ^, anchoring at the beginning of line.
r = Regexp.new("^oo")|
assert
{
  r.match("oops");
  !r.match("woot");
};

// $, anchoring at the end of line.
r = Regexp.new("oo$")|
assert
{
  r.match("foo");
  !r.match("mooh");
};

// *, greedy repetition, 0 or more.
r = Regexp.new("fo*bar")|
assert
{
  r.match("fbar");
  r.match("fooooobar");
```

```

    !r.match("far");
};

// (), grouping.
r = Regexp.new("f(oo)*bar") |
assert
{
  r.match("fooobar");
  !r.match("fooobar");
};

```

- **matches**

If the latest `match` was successful, the matched groups, as delimited by parentheses in the regular expression; the first element being the whole match. Otherwise, the empty list. See also '`[]`'.

```

var re = Regexp.new("[a-zA-Z0-9._]+@[a-zA-Z0-9._]+");
assert
{
  re.match("Someone <someone@somewhere.com>");
  re.matches == ["someone@somewhere.com", "someone", "somewhere.com"];

  "does not match" not in re;
  re.matches == [];
};

```

urbiscript
Session

- `[](n)`

Same as `this.matches[n]`.

```

var d = Regexp.new("(1+)(2+)(3+)") |
assert
{
  "01223334" in d;
  d[0] == "122333";
  d[1] == "1";
  d[2] == "22";
  d[3] == "333";
};
d[4];
[00000009:error] !!! []: out of bound index: 4

```

urbiscript
Session

24.56 Semaphore

Semaphore are useful to limit the number of access to a limited number of resources.

24.56.1 Prototypes

- `Object`

24.56.2 Construction

A `Semaphore` can be created with as argument the number of processes allowed to enter critical sections at the same time.

```

Semaphore.new(1);
[00000000] Semaphore_0x8c1e80

```

urbiscript
Session

24.56.3 Slots

- **criticalSection(function() { code })**

Put the piece of *code* inside a critical section which can be executed simultaneously at most the number of time given at the creation of the **Semaphore**. This method is similar to a call to **acquire** and a call to **release** when the code ends by any means.

urbiscript
Session

```
{
    var s = Semaphore.new(1);
    for& (var i : [0, 1, 2, 3])
    {
        s.criticalSection(function () {
            echo("start " + i);
            echo("end " + i);
        })
    }
};

[00000000] *** start 0
[00000000] *** end 0
[00000000] *** start 1
[00000000] *** end 1
[00000000] *** start 2
[00000000] *** end 2
[00000000] *** start 3
[00000000] *** end 3


{
    var s = Semaphore.new(2);
    for& (var i : [0, 1, 2, 3])
    {
        s.criticalSection(function () {
            echo("start " + i);

            // Illustrate that processes can be intertwined
            sleep(i * 100ms);

            echo("end " + i);
        })
    }
};

[00000000] *** start 0
[00000000] *** start 1
[00000000] *** end 0
[00000000] *** start 2
[00000000] *** end 1
[00000000] *** start 3
[00000000] *** end 2
[00000000] *** end 3
```

- **acquire**

Wait to enter a critical section delimited by the execution of **acquire** and **release**. Enter the critical section when the number of processes inside it goes below the maximum allowed.

- **p**

Historical synonym for **acquire**.

- **release**

Leave a critical section delimited by the execution of **acquire** and **release**.

urbiscript
Session

```
{
    var s = Semaphore.new(1);
    for& (var i : [0, 1, 2, 3])
```

```

    {
        s.acquire;
        echo("start " + i);
        echo("end " + i);
        s.release;
    }
};

[00000000] *** start 0
[00000000] *** end 0
[00000000] *** start 1
[00000000] *** end 1
[00000000] *** start 2
[00000000] *** end 2
[00000000] *** start 3
[00000000] *** end 3

```

- **v**
Historical synonym for `release`.

24.57 Server

A `Server` can listen to incoming connections. See `Socket` for an example.

24.57.1 Prototypes

- `Object`

24.57.2 Construction

A `Server` is constructed with no argument. At creation, a new `Server` has its own slot connection. This slot is an event that is launched when a connection establishes.

```

var s = Server.new|
s.localSlotNames;
[00000001] ["connection"]

```

urbiscript
Session

24.57.3 Slots

- `connection`

The event launched at each incoming connection. This event is launched with one argument: the socket of the established connection. This connection uses the same `IoService` as the server.

```

at (s.connection?(var socket))
{
    // This code is run at each connection. 'socket' is the incoming
    // connection.
};

```

urbiscript
Session

- `getIoService`

Return the `IoService` used by this socket. Only the default `IoService` is automatically polled.

- `host`

The host on which `this` is listening. Raise an error if `this` is not listening.

```

Server.host;
[00000003:error] !!! host: server not listening

```

urbiscript
Session

- `listen(host, port)`

Listen incoming connections with `host` and `port`.

- `port`

The port on which `this` is listening. Raise an error if `this` is not listening.

urbiscript
Session

```
Server.port;
[00000004:error] !!! port: server not listening
```

- `sockets`

The list of the sockets created at each incoming connection.

24.58 Singleton

A `singleton` is a prototype that cannot be cloned. All prototypes derived of `Singleton` are also singletons.

24.58.1 Prototypes

- `Object`

24.58.2 Construction

To be a singleton, the object must have `Singleton` as a prototype. The common way to do this is `var s = Singleton.new`, but this does not work : `s` is not a new singleton, it is the `Singleton` itself since it cannot be cloned. There are two other ways:

urbiscript
Session

```
// Defining a new class and specifying Singleton as a parent.
class NewSingleton1: Singleton
{
    var asString = "NewSingleton1";
}
var s1 = NewSingleton1.new;
[00000001] NewSingleton1
assert(s1 === NewSingleton1);
assert(NewSingleton1 !== Singleton);

// Create a new Object and set its prototype by hand.
var NewSingleton2 = Object.new|
var NewSingleton2.asString = "NewSingleton2"|
NewSingleton2.protos = [Singleton]|
var s2 = NewSingleton2.new;
[00000001] NewSingleton2
assert(s2 === NewSingleton2);
assert(NewSingleton2 !== Singleton);
```

24.58.3 Slots

- `clone`

Return `this`.

- `'new'`

Return `this`.

24.59 Socket

A `Socket` can manage asynchronous input/output network connections.

24.59.1 Example

The following example demonstrates how both the `Server` and `Socket` object work.

This simple example will establish a dialog between `server` and `client`. The following object, `Dialog`, contains the script of this exchange. It is put into `Global` so that both the server and client can read it. `Dialog.reply(var s)` returns the reply to a message `s`.

```
urbiscript
Session

class Global.Dialog
{
    var lines =
    [
        "Hi!",
        "Hey!",
        "Hey you doin'?",
        "Whazaaa!",
        "See ya.",
    ];
}

function reply(var s)
{
    for (var i: lines.size - 1)
        if (s == lines[i])
            return lines[i + 1];
    "off";
}
};
```

The server, an instance of `Server`, expects incoming connections, notified by the socket's `connection?` event. Once the connection establish, it listens to the `socket` for incoming messages, notified by the `received?` event. Its reaction to this event is to send the following line of the dialog. At the end of the dialog, the socket is disconnected.

```
urbiscript
Session

var server =
do (Server.new)
{
    at (connection?(var socket))
        at (socket.received?(var data))
    {
        var reply = Dialog.reply(data);
        socket.write(reply);
        echo("server: " + reply);
        if (reply == "off")
            socket.disconnect;
    };
};
```

The client, an instance of `Socket` expects incoming messages, notified by the `received?` event. Its reaction is to send the following line of the dialog.

```
urbiscript
Session

var client =
do (Socket.new)
{
    at (received?(var data))
    {
        var reply = Dialog.reply(data);
        write(reply);
        echo("client: " + reply);
    };
};
```

As of today, urbiscript's socket machinery requires to be regularly polled.

```
urbiscript
Session

every (100ms)
    Socket.poll,
```

The server is then activated, listening to incoming connections on a port that will be chosen by the system among the free ones.

urbiscript
Session

```
server.listen("localhost", "0");
clog << "connecting to %s:%s" % [server.host, server.port];
```

The client connects to the server, and initiates the dialog.

urbiscript
Session

```
client.connect(server.host, server.port);
echo("client: " + Dialog.lines[0]);
client.write(Dialog.lines[0]);
[00000003] *** client: Hi!
```

Because this dialog is asynchronous, the easiest way to wait for the dialog to finish is to wait for the `disconnected?` event.

urbiscript
Session

```
waituntil(client.disconnected?);
[00000004] *** server: Hey!
[00000005] *** client: Hey you doin'?
[00000006] *** server: Whazaaa!
[00000007] *** client: See ya.
[00000008] *** server: off
```

24.59.2 Prototypes

- [Object](#)

24.59.3 Construction

A `Socket` is constructed with no argument. At creation, a new `Socket` has four own slots: `connected`, `disconnected`, `error` and `received`.

urbiscript
Session

```
var s = Socket.new|
```

24.59.4 Slots

- `connect(host, port)`

Connect `this` to `host` and `port`. The `port` can be either an integer, or a string that denotes symbolic ports, such as `"smtp"`, or `"ftp"` and so forth.

- `connected`

Event launched when the connection is established.

- `connectSerial(device, baudRate)`

Connect `this` to the serial port `device`, with given `baudRate`.

- `disconnect`

Close the connection.

- `disconnected`

Event launched when a disconnection happens.

- `error`

Event launched when an error happens. This event is launched with the error message in argument. The event `disconnected` is also always launched.

- `getIoService`

Return the `IoService` used by this socket. Only the default `IoService` is automatically polled.

- **host**
The remote host of the connection.
- **isConnected**
Whether `this` is connected.
- **localhost**
The local host of the connection.
- **localPort**
The local port of the connection.
- **poll**
Call `getIoService.poll()`. This method is called regularly every `pollInterval` on the `Socket` object. You do not need to call this function on your sockets unless you use your own `IoService`.
- **pollInterval**
Each `pollInterval` amount of time, `poll` is called. If `pollInterval` equals zero, `poll` is not called.
- **port**
The remote port of the connection.
- **received**
Event launched when `this` has received data. The data is given by argument to the event.
- **write(*data*)**
Sends `data` through the connection.
- **syncWrite(*data*)**
Similar to `write`, but forces the operation to complete synchronously. Synchronous and asynchronous write operations cannot be mixed.

24.60 StackFrame

This class is meant to record backtrace (see `Exception.backtrace`) information.

For convenience, all snippets of code are supposed to be run after these function definitions. In this code, the `getStackFrame` function is used to get the first `StackFrame` of an exception backtrace. Backtrace of `Exception` are filled with `StackFrames` when they are thrown.

```
//#push 1 "foo.u"
function inner () { throw Exception.new("test") }|;
function getStackFrame()
{
  try
  {
    inner
  }
  catch(var e)
  {
    e.backtrace[0]
  };
}|;
//pop
```

urbiscript
Session

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

24.60.1 Construction

`StackFrame` are not made to be manually constructed. The initialization function expect 2 arguments, which are the name of the called function and the `Location` from which it has been called.

urbiscript
Session

```
StackFrame.new("inner",
  Location.new(
    Position.new("foo.u", 7, 5),
    Position.new("foo.u", 7, 10)
  )
);
[00000001] foo.u:7.5-9: inner
```

24.60.2 Slots

- `name`

`String`, representing the name of the called function.

urbiscript
Session

```
getStackFrame.name;
[00000002] "inner"
```

- `location`

`Location` of the function call.

urbiscript
Session

```
getStackFrame.location;
[00000003] foo.u:7.5-9
```

- `asString`

Clean display of the call location.

urbiscript
Session

```
getStackFrame;
[00000004] foo.u:7.5-9: inner
```

24.61 Stream

This is used to factor code between `InputStream` and `OutputStream`.

24.61.1 Prototypes

- `Object`

24.61.2 Construction

Streams are not meant to be built, rather, use `InputStream` or `OutputStream`.

When a stream (`OutputStream` or `InputStream`) is opened on a File, that File cannot be removed. On Unix systems, this is handled gracefully (the references to the file are removed, but the content is still there for the streams that were already bound to this file); so in practice, the File appears to be removable. On Windows, the File cannot be removed at all. Therefore, do not forget to close the streams you opened.

24.61.3 Slots

- `close`

Flush the buffers, close the stream, return void. Raise an error if the file is closed.

urbiscript
Session

```
{
    var i = InputStream.new(File.create("file.txt"));
    assert(i.close.isVoid);
    i.close;
};

[00000001:error] !!! close: stream is closed

{
    var o = OutputStream.new(File.create("file.txt"));
    assert(o.close.isVoid);
    o.close;
};

[00000002:error] !!! close: stream is closed
```

24.62 String

A *string* is a sequence of characters.

24.62.1 Prototypes

- Comparable
- Orderable
- RangeIterable

24.62.2 Construction

Fresh Strings can easily be built using the literal syntax. Several escaping sequences (the traditional ones and urbiscript specific ones) allow to insert special characters. Consecutive string literals are merged together. See [Section 23.1.6.6](#) for details and examples.

A null String can also be obtained with `String's new` method.

```
String.new == "";
String == "";
"123".new == "123";
```

Assertion Block

24.62.3 Slots

- `asFloat`

If the whole content of `this` is a number, return it as a `Float`, otherwise raise an error.

```
assert
{
    "23".asFloat == 23;
    "23.03".asFloat == 23.03;
};

"123abc".asFloat;
[00000001:error] !!! asFloat: cannot convert to float: "123abc"
```

urbiscript Session

- `asList`

Return a List of one-letter Strings that, concatenated, equal `this`. This allows to use `for` to iterate over the string.

```
assert("123".asList == ["1", "2", "3"]);
for (var v : "123")
    echo(v);
```

urbiscript Session

```
[00000001] *** 1
[00000001] *** 2
[00000001] *** 3
```

- **asPrintable**

Return `this` as a literal (escaped) string.

Assertion Block

```
"foo".asPrintable == "\"foo\"";
"foo".asPrintable.asPrintable == "\"\\\\\"foo\\\\\"\"";
```

- **asString**

Return `this`.

Assertion Block

```
"\\\"foo\\\".asString == \"\\\"foo\\\"\";
```

- **closest(set)**

Return the string in `set` that is the closest (in the sense of `distance`) to `this`. If there is no convincing match, return `nil`.

Assertion Block

```
"foo".closest(["foo", "baz", "qux", "quux"]) == "foo";
"bar".closest(["foo", "baz", "qux", "quux"]) == "baz";
"FOO".closest(["foo", "bar", "baz"])      == "foo";
"qux".closest(["foo", "bar", "baz"])      == nil;
```

- **distance(other)**

Return the [Damerau-Levenshtein distance](#) between `this` and `other`. The more alike the strings are, the smaller the distance is.

Assertion Block

```
"foo".distance("foo") == 0;
"bar".distance("baz") == 1;
"foo".distance("bar") == 3;
```

- **empty**

Whether this is the empty string.

Assertion Block

```
"".empty;
!"x".empty;
```

- **fresh**

Return a String that has never been used as an identifier, prefixed by `this`. It can safely be used with `Object.setSlot` and so forth.

Assertion Block

```
String.fresh == "_5";
"foo".fresh == "foo_6";
```

- **fromAscii(v)**

The character corresponding to the integer `v` according to the ASCII coding. See also `toAscii`.

Assertion Block

```
String.fromAscii( 97) == "a";
String.fromAscii( 98) == "b";
String.fromAscii(0xFF) == "\\xff";
[0, 1, 2, 254, 255]
  .map(function (var v) { String.fromAscii(v) })
  .map(function (var v) { v.toAscii })
  == [0, 1, 2, 254, 255];
```

- **hash**

A `Hash` object corresponding to this string value. Two string hashes are equal if the string are equal. See `Object.hash`.

```
"".hash.isA(Hash);
"foo".hash == "foo".hash;
"foo".hash != "bar".hash;
```

Assertion Block

- Character handling functions

Here is a map of how the original 127-character ASCII set is considered by each function (a • indicates that the function returns true if all characters of `this` are on the row).

ASCII values	Characters	iscntrl	isspace	isupper	islower	isalpha	isdigit	isxdigit	isalnum	ispunct	isgraph	print
0x00 .. 0x08		•										
0x09 .. 0x0D	\t, \f, \v, \n, \r	•	•									
0x0E .. 0x1F		•										
0x20	space (' ')		•									•
0x21 .. 0x2F	! "#\$%&'()*+, - . /								•	•	•	
0x30 .. 0x39	0-9					•	•	•		•	•	
0x3a .. 0x40	: ; <= > ? @								•	•	•	
0x41 .. 0x46	A-F		•		•		•	•		•	•	
0x47 .. 0x5A	G-Z		•		•			•		•	•	
0x5B .. 0x60	[\] ^ { } _ `								•	•	•	
0x61 .. 0x66	a-f			•	•		•	•		•	•	
0x67 .. 0x7A	g-z			•	•			•		•	•	
0x7B .. 0x7E	{ } ~								•	•	•	
0x7F	(DEL)	•										

Assertion Block

```
"".isDigit;
"0123456789".isDigit;
!"a".isDigit;

"".isLower;
"lower".isLower;
!"Not Lower".isLower;

"".isUpper;
"UPPER".isUpper;
!"Not Upper".isUpper;
```

- **join(*list*, *prefix*, *suffix*)**

Glue the result of `asString` applied to the members of *list*, separated by `this`, and embedded in a pair *prefix/suffix*.

```
"| ".join([1, 2, 3], "( ", ")") == "(1|2|3)";
", ".join([1, [2], "3"], "[", "]") == "[1, [2], 3]";
```

Assertion Block

- **length**

The number of characters in the string. Currently, this is a synonym of `size`.

```
"foo".length == 3;
"".length == 0;
```

Assertion Block

- **replace(*from*, *to*)**

Replace every occurrence of the string *from* in **this** by *to*, and return the result. **this** is not modified.

Assertion Block

```
"Hello, World!".replace("Hello", "Bonjour")
    .replace("World!", "Monde !") ==
"Bonjour, Monde !";
```

- **size**

The size of the string.

Assertion Block

```
"foo".size == 3;
"".size == 0;
```

- **split(*sep* = [" ", "\t", "\n", "\r"], *lim* = -1, *keepSep* = false, *keepEmpty* = true)**

Split **this** on the separator *sep*, in at most *lim* components, which include the separator if *keepSep*, and the empty components of *keepEmpty*. Return a list of strings.

The separator, *sep*, can be a string.

Assertion Block

```
"a,b;c".split(",") == ["a", "b;c"];
"a,b;c".split(";") == ["a,b", "c"];
"foobar".split("x") == ["foobar"];
"foobar".split("ob") == ["fo", "ar"];
```

It can also be a list of strings.

Assertion Block

```
"a,b;c".split([" ", ";"]) == ["a", "b", "c"];
```

By default splitting is performed on white-spaces:

Assertion Block

```
" abc  def\tghi\n".split == ["abc", "def", "ghi"];
```

Splitting on the empty string stands for splitting between each character:

Assertion Block

```
"foobar".split("") == ["f", "o", "o", "b", "a", "r"];
```

The limit *lim* indicates a maximum number of splits that can occur. A negative number corresponds to no limit:

Assertion Block

```
"a:b:c".split(":", 1) == ["a", "b:c"];
"a:b:c".split(":", -1) == ["a", "b", "c"];
```

keepSep indicates whether to keep delimiters in the result:

Assertion Block

```
"aaa:bbb;ccc".split([":", ";"], -1, false) == ["aaa", "bbb", "ccc"];
"aaa:bbb;ccc".split([":", ";"], -1, true) == ["aaa", ":", "bbb", ";", "ccc"];
```

keepEmpty indicates whether to keep empty elements:

Assertion Block

```
"foobar".split("o") == ["f", "", "bar"];
"foobar".split("o", -1, false, true) == ["f", "", "bar"];
"foobar".split("o", -1, false, false) == ["f", "bar"];
```

- **toAscii**

Convert the first character of **this** to its integer value in the ASCII coding. See also [fromAscii](#).

Assertion Block

```

    "a".toAscii == 97;
    "b".toAscii == 98;
"\ufffd".toAscii == 0xff;
"Hello, World!\n"
.asList
.map(function (var v) { v.toAscii })
.map(function (var v) { String.fromCharCode(v) })
.join
== "Hello, World!\n";

```

- **toLowerCase**

Make lower case every upper case character in `this` and return the result. `this` is not modified.

```
"Hello, World!".toLowerCase == "hello, world!";
```

Assertion Block

- **toUpperCase**

Make upper case every lower case character in `this` and return the result. `this` is not modified.

```
"Hello, World!".toUpperCase == "HELLO, WORLD!";
```

Assertion Block

- **'==' (*that*)**

Whether `this` and `that` are the same string.

```

"" == "";
!("") != "";

"0" == "0";
!("0" != "0");
!("0" == "1");
"0" != "1";
!("1" == "0");
"1" != "0";

```

Assertion Block

- **'%' (*args*)**

It is an equivalent of `Formatter.new(this) % args`. See [Formatter](#).

```
"%s + %s = %s" % [1, 2, 3] == "1 + 2 = 3";
```

Assertion Block

- **'*' (*n*)**

Concatenate `this`*n* times.

```

"foo" * 0 == "";
"foo" * 1 == "foo";
"foo" * 3 == "foofoofoo";

```

Assertion Block

- **'+' (*other*)**

Concatenate `this` and `other.asString`.

```

"foo" + "bar" == "foobar";
"foo" + "" == "foo";
"foo" + 3 == "foo3";
"foo" + [1, 2, 3] == "foo[1, 2, 3]";

```

Assertion Block

- **'<' (*other*)**

Whether `this` is lexicographically before `other`, which must be a String.

```

"" < "a";
!("a" < "");
"a" < "b";
!("a" < "a");

```

Assertion Block

- `'[]' (from)`
`'[]' (from, to)`

Return the sub-string starting at *from*, up to and not including *to* (which defaults to *to* + 1).

Assertion Block

```
"foobar"[0, 3] == "foo";
"foobar"[0] == "f";
```

- `'[]=' (from, other)`
`'[]=' (from, to, other)`

Replace the sub-string starting at *from*, up to and not including *to* (which defaults to *to* + 1), by *other*. Return *other*.

Beware that this routine is imperative: it changes the value of `this`.

urbiscript Session

```
var s1 = "foobar" | var s2 = s1 |
assert((s1[0, 3] = "quux") == "quux");
assert(s1 == "quuxbar");
assert(s2 == "quuxbar");
assert((s1[4, 7] = "") == "");
assert(s2 == "quux");
```

24.63 System

Details on the architecture the Urbi server runs on.

24.63.1 Prototypes

- `Object`

24.63.2 Slots

- `_exit(status)`

Shut the server down brutally: the connections are not closed, and the resources are not explicitly released (the operating system reclaims most of them: memory, file descriptors and so forth). Architecture dependent.

- `aliveJobs`

The number of detached routines currently running.

urbiscript Session

```
{
    var nJobs = aliveJobs;
    for (var i: [1s, 2s, 3s])
        detach({sleep(i)});
    sleep(0.5s);
    assert(aliveJobs - nJobs == 3);
    sleep(1s);
    assert(aliveJobs - nJobs == 2);
    sleep(1s);
    assert(aliveJobs - nJobs == 1);
    sleep(1s);
    assert(aliveJobs - nJobs == 0);
};
```

- `arguments`

The list of the command line arguments passed to the user script. This is especially useful in scripts.

Shell Session

```
$ cat >echo <<EOF
#!/usr/bin/env urbi
System.arguments;
shutdown;
EOF
$ chmod +x echo
$ ./echo 1 2 3
[00000172] [1, 2, 3]
$ ./echo -x 12 -v "foo"
[00000172] [-x, 12, -v, foo]
```

- **'assert'(*assertion*)**

Unless `ndebug` is true, throw an error if *assertion* is not verified. See also the assertion support in urbiscript, [Section 23.9](#).

```
'assert'(true);
'assert'(42);
'assert'(1 == 1 + 1);
[00000002:error] !!! failed assertion: 1.'=='(1.'+'(1))
```

urbiscript
Session

- **assert_(*assertion*, *message*)**

If *assertion* does not evaluate to true, throw the failure *message*.

```
assert_(true,      "true failed");
assert_(42,        "42 failed");
assert_(1 == 1 + 1, "one is not two");
[00000001:error] !!! failed assertion: one is not two
```

urbiscript
Session

- **assert_op(*operator*, *lhs*, *rhs*)**

Deprecated, use `assert` instead, see [Section 23.9](#).

- **backtrace**

Display the call stack on the channel `backtrace`.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

```
//#push 100 "foo.u"
function innermost () { backtrace }|;
function inner ()   { innermost }|;
function outer ()  { inner }|;
function outermost () { outer }|;
outermost;
[00000013:backtrace] innermost (foo.u:101.25-33)
[00000014:backtrace] inner (foo.u:102.25-29)
[00000015:backtrace] outer (foo.u:103.25-29)
[00000016:backtrace] outermost (foo.u:104.1-9)
//#pop
```

urbiscript
Session

- **cycle**

The number of execution cycles since the beginning.

This feature is experimental. It might be changed in the future. Feedback on its use would be appreciated.

```
{
  var first = cycle ; var second = cycle ;
  assert(first + 1 == second);
  first = cycle | second = cycle ;
  assert(first == second);
};
```

urbiscript
Session

- **`eval(source, target = this)`**

Evaluate the urbiscript `source`, and return its result. See also [loadFile](#). The `source` must be complete, yet the terminator (e.g., ‘;’) is not required.

Assertion Block

```
eval("1+2") == 1+2;
eval("\\"x\" * 10") == "x" * 10;
eval("eval(\"1\")") == 1;
eval("{ var x = 1; x + x; }") == 2;
```

The evaluation is performed in the context of the current object (`this`) or `target` if specified. In particular, to create local variables, create scopes.

Assertion Block

```
// Create a slot in the current object.
eval("var a = 23;") == 23;
this.a == 23;

eval("var a = 3", Global) == 3;
Global.a == 3;
```

Exceptions are thrown on error (including syntax errors).

urbiscript Session

```
eval("1/0");
[00008316:error] !!! 1.1-3: /: division by 0
[00008316:error] !!!      called from: eval
try
{
  eval("1/0")
}
catch (var e)
{
  assert
  {
    e.isA(Exception.Primitive);
    e.location.asString  == "1.1-3";
    e.routine           == "/";
    e.message            == "division by 0";
  }
};
```

Warnings are reported.

urbiscript Session

```
eval("new Object");
[00001388:warning] !!! 1.1-10: 'new Obj(x)' is deprecated, use 'Obj.new(x)'
[00001388:warning] !!!      called from: eval
[00001388] Object_0x1001b2320
```

Nested calls to `eval` behave as expected. The locations in the inner calls refer to the position inside the evaluated string.

urbiscript Session

```
eval("/");
[00001028:error] !!! 1.1: syntax error: unexpected /
[00001028:error] !!!      called from: eval

eval("eval(\"/\")");
[00001032:error] !!! 1.1: syntax error: unexpected /
[00001032:error] !!!      called from: 1.1-9: eval
[00001032:error] !!!      called from: eval

eval("eval(\"eval("\\\\\"\\\\\\\")\")");
[00001035:error] !!! 1.1: syntax error: unexpected /
[00001035:error] !!!      called from: 1.1-9: eval
[00001035:error] !!!      called from: 1.1-19: eval
[00001035:error] !!!      called from: eval
```

- **getenv(*name*)**

The value of the environment variable *name* as a [String](#) if set, [nil](#) otherwise. See also [setenv](#) and [unsetenv](#).

```
getenv("UndefinedEnvironmentVariable").isNil;
!getenv("PATH").isNil;
```

Assertion Block

- **load(*file*, *target* = [this](#))**

Look for *file* in the Urbi path ([Section 22.1](#)), and load it in the context of *target*. See also [loadFile](#). Throw a [Exception.FileNotFound](#) error if the file cannot be found. Return the last value of the file.

```
// Create the file ``123.u'' that contains exactly ``var t = 123;''.
File.save("123.u", "var t = 123;");
assert
{
    load("123.u") == 123;
    this.t == 123;

    load("123.u", Global) == 123;
    Global.t == 123;
};
```

urbiscript Session

- **loadFile(*file*, *target* = [this](#))**

Load the urbiscript file *file* in the context of *target*. Behaves like [eval](#) applied to the content of *file*. Throw a [Exception.FileNotFound](#) error if the file cannot be found. Return the last value of the file.

```
// Create the file ``123.u'' that contains exactly ``var y = 123;''.
File.save("123.u", "var y = 123;");
assert
{
    loadFile("123.u") == 123;
    this.y == 123;

    loadFile("123.u", Global) == 123;
    Global.y == 123;
};
```

urbiscript Session

- **loadLibrary(*library*)**

Load the library *library*, to be found in [UObject.searchPath](#). The *library* may be a [String](#) or a [Path](#). The C++ symbols are made available to the other C++ components. See also [loadModule](#).

- **loadModule(*module*)**

Load the UObject*module*. Same as [loadLibrary](#), except that the low-level C++ symbols are not made “global” (in the sense of the shared library loader).

- **lobbies**

Bounce to [Lobby.instances](#).

- **lobby**

Bounce to [Lobby.lobby](#).

- **maybeLoad(*file*, *channel* = [Channel.null](#))**

Look for *file* in the Urbi path ([Section 22.1](#)). If the file is found announce on *Channel* that *file* is about to be loaded, and load it.

urbiscript Session

```
// Create the file "123.u" that contains exactly "123;".
File.save("123.u", "123;");
assert
{
    maybeLoad("123.u") == 123;
    maybeLoad("u.123").isVoid;
};
```

- **ndebug**

If true, do not evaluate the assertions. See [Section 23.9](#).

urbiscript
Session

```
function one() { echo("called!"); 1 }|;
assert(!System.ndebug);

assert(one);
[00000617] *** called!

// Beware of copy-on-write.
System.ndebug = true|;
assert(one);

System.ndebug = false|;
assert(one);
[00000622] *** called!
```

- **PackageInfo**

See [System.PackageInfo](#).

- **period**

The *period* of the Urbi kernel. Influences the trajectories ([TrajectoryGenerator](#)), and the **UObject** monitoring. Defaults to 20ms.

Assertion
Block

```
System.period == 20ms;
```

- **Platform**

See [System.Platform](#).

- **profile(function)**

Compute some measures during the execution of *function* and return the results as a **Profile** object. A **Profile** details information about time, function calls and scheduling.

- **programName**

The path under which the Urbi process was called. This is typically ‘.../urbi’ ([Section 22.3](#)) or ‘.../urbi-launch’ ([Section 22.5](#)).

Assertion
Block

```
Path.new(System.programName).basename
in ["urbi", "urbi.exe", "urbi-launch", "urbi-launch.exe"];
```

- **reboot**

Restart the Urbi server. Architecture dependent.

- **redefinitionMode**

Switch the current job in redefinition mode until the end of the current scope. While in redefinition mode, setSlot on already existing slots will overwrite the slot instead of erring.

urbiscript
Session

```
var Global.x = 0;
[00000001] 0
{
    System.redefinitionMode;
```

```
// Not an error
var Global.x = 1;
echo(Global.x);
};

[00000002] *** 1
// redefinitionMode applies only to the scope.
var Global.x = 0;
[00000003:error] !!! slot redefinition: x
```

- **requireFile(*file*, *target*)**

Load *file* in the context of *target* if it was not loaded before (with `load` or `requireFile`). Unlike `load`, `requireFile` always returns `void`. If *file* is being loaded concurrently `requireFile` waits until the loading is finished.

```
// Create the file "test.u" that echoes a string.
File.save("test1.u", "echo(\"test 1\"); 1;");
requireFile("test1.u");
[00000001] *** test 1
requireFile("test1.u");
// File is not re-loaded

File.save("test2.u", "echo(\"test 2\"); 2;");
load("test2.u");
[00000004] *** test 2
[00000004] 2
requireFile("test2.u");
load("test2.u");
[00000006] *** test 2
[00000006] 2
```

urbiscript
Session

The *target* is not taken into account to check whether the file has already been loaded: if you require twice the same file with two different targets, it will be loaded only for the first.

```
requireFile("test2.u", Global);
```

urbiscript
Session

- **resetStats**

Reinitialize the `stats` computation.

```
0 < System.stats["cycles"];
System.resetStats.isVoid;
1 == System.stats["cycles"];
```

Assertion
Block

- **scopeTag**

Bounce to `Tag.scope`.

- **searchFile(*file*)**

Look for *file* in the `searchPath` and return its `Path`. Throw a `Exception.FileNotFound` error if the file cannot be found.

```
// Create the file "123.u" that contains exactly "123;".
File.save("123.u", "123;");
assert
{
    searchFile("123.u") == Path.cwd / Path.new("123.u");
};
```

urbiscript
Session

- **searchPath**

The Urbi path (i.e., the directories where the urbiscript files are looked for, see [Section 22.1](#)) as a `List` of `Paths`.

Assertion
Block

```
System.searchPath.isA(List);
System.searchPath[0].isA(Path);
```

- **setenv(*name*, *value*)**

Set the environment variable *name* to *value*.asString, and return this value. See also [getenv](#) and [unsetenv](#).

Windows Issues

Under Windows, setting to an empty value is equivalent to making undefined.

Assertion Block

```
setenv("MyVar", 12) == "12";
getenv("MyVar") == "12";

// A child process that uses the environment variable.
System.system("exit $MyVar") >> 8 ==
    {if (Platform.isWindows) 0 else 12};
setenv("MyVar", 23) == "23";
System.system("exit $MyVar") >> 8 ==
    {if (Platform.isWindows) 0 else 23};

// Defining to empty is defining, unless you are on Windows.
setenv("MyVar", "") == "";
getenv("MyVar").isNil == Platform.isWindows;
```

- **shiftedTime**

Return the number of seconds elapsed since the Urbi server was launched. Contrary to [time](#), time spent in frozen code is not counted.

Assertion Block

```
{ var t0 = shiftedTime | sleep(1s) | shiftedTime - t0 }.round ~= 1;

1 ==
{
    var t = Tag.new();
    var t0 = time();
    var res;
    t: { sleep(1s) | res = shiftedTime - t0 },
    t.freeze();
    sleep(1s);
    t.unfreeze();
    sleep(1s);
    res.round;
};
```

- **shutdown**

Have the Urbi server shut down. All the connections are closed, the resources are released. Architecture dependent.

- **sleep(*duration* = inf)**

Suspend the execution for *duration* seconds. No CPU cycle is wasted during this wait. If no *duration* is given the execution is suspended indefinitely.

Assertion Block

```
(time - {sleep(1s); time}).round == -1;
```

- **spawn(*function*, *clear*)**

Run the *function*, with fresh tags if *clear* is true, otherwise under the control of the current tags. Return the spawn [Job](#).

```

var jobs = []|;
var res = []|;
for (var i : [0, 1, 2])
{
    jobs << System.spawn(closure () { res << i; res << i },
                           true) |
    if (i == 2)
        break
}||
jobs;
[00009120] [Job<shell_11>, Job<shell_12>, Job<shell_13>]
// Wait for the jobs to be done.
jobs.each (function (var j) { j.waitForTermination });
assert (res == [0, 1, 0, 2, 1, 2]);

```

```

jobs = []|;
res = []|;
for (var i : [0, 1, 2])
{
    jobs << System.spawn(closure () { res << i; res << i },
                           false) |
    if (i == 2)
        break
}||
jobs;
[00009120] [Job<shell_14>, Job<shell_15>, Job<shell_16>]
// Give some time to get the output of the detached expressions.
sleep(100ms);
assert (res == [0, 1, 0]);

```

urbiscript
Session

- **stats**

Return a [Dictionary](#) containing information about the execution cycles of Urbi. This is an internal feature made for developers, it might be changed without notice. See also [resetStats](#).

```

var stats = System.stats|;
assert
{
    stats.isA(Dictionary);
    stats.keys.sort == ["cycles",
                        "cyclesMin", "cyclesMean", "cyclesMax",
                        "cyclesVariance", "cyclesStdDev"].sort;
    0 < stats["cycles"];
    stats["cyclesMin"]  <= stats["cyclesMean"];
    stats["cyclesMean"] <= stats["cyclesMax"];
};

```

urbiscript
Session

- **system(*command*)**

Ask the operating system to run the *command*. This is typically used to start new processes. The exact syntax of *command* depends on your system. On Unix systems, this is typically ‘/bin/sh’, while Windows uses ‘command.exe’.

Return the exit status.

Windows Issues

Under Windows, the exit status is always 0.

```

System.system("exit 0") == 0;
System.system("exit 23") >> 8
    == { if (System.Platform.isWindows) 0 else 23 };

```

Assertion
Block

- **time**

Return the number of seconds elapsed since the Urbi server was launched. In presence of a frozen **Tag**, see also **shiftedTime**.

Assertion Block

```
{ var t0 = time | sleep(1s) | time - t0 }.round ~= 1;

2 ==
{
    var t = Tag.new();
    var t0 = time;
    var res;
    t: { sleep(1s) | res = time - t0 },
    t.freeze;
    sleep(1s);
    t.unfreeze;
    sleep(1s);
    res.round;
};
```

- **timeReference**

The “origin of time” of this run of Urbi, as a **Date**. It is a constant during the run. Basically, **System.time** is about **Date.now** - **System.timeReference**. See also **time** and **Date.now**.

urbiscript Session

```
var t1 = System.timeReference;
sleep(1s);
var t2 = System.timeReference;
assert
{
    t1 == t2;
    t1.isA(Date);
    (Date.now - (System.timeReference + System.time)) < 0.5s;
};
```

- **unsetenv(name)**

Undefine the environment variable **name**, return its previous value. See also **getenv** and **setenv**.

Assertion Block

```
setenv("MyVar", 12) == "12";
!getenv("MyVar").isNil;
unsetenv("MyVar") == "12";
getenv("MyVar").isNil;
```

- **version**

The version of Urbi SDK. A string composed of two or more numbers separated by periods: “**2.7.5**”.

Assertion Block

```
System.version in Regexp.new("\\d+(\\.\\d+)+");
```

24.64 System.PackageInfo

Information about Urbi SDK and its components.

24.64.1 Prototypes

- **Object**

24.64.2 Slots

- **copyrightHolder**

The Urbi SDK copyright holder.

```
System.PackageInfo.copyrightHolder == "Gostai S.A.S.;"
```

Assertion
Block

- **copyrightYears**

The Urbi SDK copyright years.

```
System.PackageInfo.copyrightYears == "2004-2012";
```

Assertion
Block

24.65 System.Platform

A description of the platform (the computer) the server is running on.

24.65.1 Prototypes

- [Object](#)

24.65.2 Slots

- **host**

The type of system Urbi SDK runs on. Composed of the CPU, the vendor, and the OS.

```
System.Platform.host ==
    "%s-%s-%s" % [System.Platform.hostCpu,
                    System.Platform.hostVendor,
                    System.Platform.hostOs];
```

Assertion
Block

- **hostAlias**

The name of the system Urbi SDK runs on as the person who compiled it decided to name it. Typically empty, it is fragile to depend on it.

```
System.Platform.hostAlias.isA(String);
```

Assertion
Block

- **hostCpu**

The CPU type of system Urbi SDK runs on. The following values are those for which Gostai provides binary builds.

```
System.Platform.hostCpu in ["i386", "i686", "x86_64"];
```

Assertion
Block

- **hostOs**

The OS type of system Urbi SDK runs on. For instance darwin9.8.0 or linux-gnu or mingw32.

- **hostVendor**

The vendor type of system Urbi SDK runs on. The following values are those for which Gostai provides binary builds.

```
System.Platform.hostVendor in ["apple", "pc", "unknown"];
```

Assertion
Block

- **isWindows**

Whether running under Windows.

```
System.Platform.isWindows in [true, false];
```

Assertion
Block

- `kind`

Either "POSIX" or "WIN32".

Assertion
Block

```
System.Platform.kind in ["POSIX", "WIN32"];
```

24.66 Tag

A *tag* is an object meant to label blocks of code in order to control them externally. Tagged code can be frozen, resumed, stopped... See also [Section 14.3](#).

24.66.1 Examples

24.66.1.1 Stop

To *stop* a tag means to kill all the code currently running that it labels. It does not affect "newcomers".

urbiscript
Session

```
var t = Tag.new();
var t0 = time();
t: every(1s) echo("foo"),
sleep(2.2s);
[00000158] *** foo
[00001159] *** foo
[00002159] *** foo

t.stop;
// Nothing runs.
sleep(2.2s);

t: every(1s) echo("bar"),
sleep(2.2s);
[00000158] *** bar
[00001159] *** bar
[00002159] *** bar

t.stop;
```

`Tag.stop` can be used to inject a return value to a tagged expression.

urbiscript
Session

```
var t = Tag.new();
var res;
detach(res = { t: every(1s) echo("computing") })|;
sleep(2.2s);
[00000001] *** computing
[00000002] *** computing
[00000003] *** computing

t.stop("result");
assert(res == "result");
```

Be extremely cautious, the precedence rules can be misleading: `var = tag: exp` is read as `(var = tag): exp` (i.e., defining `var` as an alias to `tag` and using it to tag `exp`), not as `var = { tag: exp }`. Contrast the following example, which is most probably an error from the user, with the previous, correct, one.

urbiscript
Session

```
var t = Tag.new("t")|;
var res;
res = t: every(1s) echo("computing"),
sleep(2.2s);
[00000001] *** computing
[00000002] *** computing
[00000003] *** computing
```

```
t.stop("result");
assert(res == "result");
[00000004:error] !!! failed assertion: res == "result" (Tag<t> != "result")
```

24.66.1.2 Block/unblock

To *block* a tag means:

- Stop running pieces of code it labels (as with `stop`).
- Ignore new pieces of code it labels (this differs from `stop`).

One can *unblock* the tag. Contrary to `freeze/unfreeze`, tagged code does not resume the execution.

```
var ping = Tag.new("ping");
ping:
  every (1s)
    echo("ping"),
  assert(!ping.blocked);
sleep(2.1s);
[00000000] *** ping
[00002000] *** ping
[00002000] *** ping

ping.block;
assert(ping.blocked);

ping:
  every (1s)
    echo("pong"),

// Neither new nor old code runs.
ping.unblock;
assert(!ping.blocked);
sleep(2.1s);

// But we can use the tag again.
ping:
  every (1s)
    echo("ping again"),
sleep(2.1s);
[00004000] *** ping again
[00005000] *** ping again
[00006000] *** ping again
```

urbiscript
Session

As with `stop`, one can force the value of stopped expressions.

```
{
  var t = Tag.new;
  var res = [];
  for (3)
    detach(res << {t: sleep});
  t.block("foo");
  res;
}
===
["foo", "foo", "foo"];
```

Assertion
Block

24.66.1.3 Freeze/unfreeze

To *freeze* a tag means holding the execution of code it labels. This applies to code already being run, and “arriving” pieces of code.

urbiscript
Session

```

var t = Tag.new();
var t0 = time();
t: every(1s) echo("time    : %.0f" % (time - t0)),
sleep(2.2s);
[00000158] *** time    : 0
[00001159] *** time    : 1
[00002159] *** time    : 2

t.freeze();
assert(t.frozen);
t: every(1s) echo("shifted: %.0f" % (shiftedTime - t0)),
sleep(2.2s);
// The tag is frozen, nothing is run.

// Unfreeze the tag: suspended code is resumed.
// Note the difference between "time" and "shiftedTime".
t.unfreeze();
assert(!t.frozen);
sleep(2.2s);
[00004559] *** shifted: 2
[00005361] *** time    : 5
[00005560] *** shifted: 3
[00006362] *** time    : 6
[00006562] *** shifted: 4

```

24.66.1.4 Scope tags

Scopes feature a `scopeTag`, i.e., a tag which will be stopped when the execution reaches the end of the current scope. This is handy to implement cleanups, however the scope was exited from.

urbiscript
Session

```

{
  var t = scopeTag;
  t: every(1s)
    echo("foo"),
    sleep(2.2s);
};

[00006562] *** foo
[00006562] *** foo
[00006562] *** foo

{
  var t = scopeTag;
  t: every(1s)
    echo("bar"),
    sleep(2.2s);
    throw 42;
};

[00006562] *** bar
[00006562] *** bar
[00006562] *** bar
[00006562:error] !!! 42
sleep(2s);

```

24.66.1.5 Enter/leave events

Tags provide two events, `enter` and `leave`, that trigger whenever flow control enters or leaves tagged statements.

urbiscript
Session

```

var t = Tag.new("t");
[00000000] Tag<t>

at (t.enter?)

```

```

echo("enter");
at (t.leave?)
echo("leave");

t: {echo("inside"); 42};
[00000000] *** enter
[00000000] *** inside
[00000000] *** leave
[00000000] 42

```

This feature is fundamental; it is a concise and safe way to ensure code will be executed upon exiting a chunk of code (like RAII in C++ or `finally` in Java). The exit code will be run no matter what the reason for leaving the block was: natural exit, exceptions, flow control statements like `return` or `break`, ...

For instance, suppose we want to make sure we turn the gas off when we're done cooking. Here is the *bad* way to do it:

```

{
  function cook()
  {
    turnGasOn();
    // Cooking code ...
    turnGasOff();
  }

  enterTheKitchen();
  cook();
  leaveTheKitchen();
}

```

urbiscript
Session

This `cook` function is wrong because there are several situations where we could leave the kitchen with gas still turned on. Consider the following cooking code:

```

{
  function cook()
  {
    turnGasOn();

    if (mealReady)
    {
      echo("The meal is already there, nothing to do!");
      // Oops ...
      return;
    };

    for (var i in recipe)
      if (i in kitchen)
        putIngredient(i)
      else
        // Oops ...
        throw Exception("missing ingredient: %s" % i);

    // ...

    turnGasOff();
  }
}

```

urbiscript
Session

Here, if the meal was already prepared, or if an ingredient is missing, we will leave the `cook` function without executing the `turnGasOff` statement, through the `return` statement or the exception. One correct way to ensure gas is necessarily turned off is:

```

{
  function cook()
}

```

urbiscript
Session

```
{
  var withGas = Tag.new("withGas");

  at (withGas.enter?)
    turnGasOn();
    // Even if exceptions are thrown or return is called,
    // the gas will be turned off.
  at (withGas.leave?)
    turnGasOff();

  withGas: {
    // Cooking code...
  }
}|;
};
```

Alternatively, the `try/finally` construct provides an elegant means to achieve the same result ([Section 23.8.4](#)).

urbiscript
Session

```
{
  function cook()
  {
    try
    {
      turnGasOn();
      // Cooking code...
    }
    finally
    {
      // Even if exceptions are thrown or return is called,
      // the gas will be turned off.
      turnGasOff();
    }
  }
};
```

24.66.1.6 Begin/end

The `begin` and `end` methods enable to monitor when code is executed. The following example illustrates the proper use of `enter` and `leave` events ([Section 24.66.1.5](#)), which are used to implement this feature.

urbiscript
Session

```
var myTag = Tag.new("myTag");
[00000000] Tag<myTag>

myTag.begin: echo(1);
[00000000] *** myTag: begin
[00000000] *** 1

myTag.end: echo(2);
[00000000] *** 2
[00000000] *** myTag: end

myTag.begin.end: echo(3);
[00000000] *** myTag: begin
[00000000] *** 3
[00000000] *** myTag: end
```

24.66.2 Construction

As any object, tags are created using `new` to create derivatives of the `Tag` object. The name is optional, it makes easier to display a tag and remember what it is.

urbiscript
Session

```
// Anonymous tag.
var t1 = Tag.new;
[00000001] Tag<tag_8>

// Named tag.
var t2 = Tag.new("cool name");
[00000001] Tag<cool name>
```

24.66.3 Slots

- **begin**

A sub-tag that prints out "tag.name: begin" each time flow control enters the tagged code. See [Section 24.66.1.6](#).

- **block(*result* = void)**

Block any code tagged by `this`. Blocked tags can be unblocked using `unblock`. If some *result* was specified, let stopped code return *result* as value. See [Section 24.66.1.2](#).

- **blocked**

Whether code tagged by `this` is blocked. See [Section 24.66.1.2](#).

- **end**

A sub-tag that prints out "tag.name: end" each time flow control leaves the tagged code. See [Section 24.66.1.6](#).

- **enter**

An event triggered each time the flow control enters the tagged code. See [Section 24.66.1.5](#).

- **freeze**

Suspend code tagged by `this`, already running or forthcoming. Frozen code can be later unfrozen using `unfreeze`. See [Section 24.66.1.3](#).

- **frozen**

Whether the tag is frozen. See [Section 24.66.1.3](#).

- **leave**

An event triggered each time flow control leaves the tagged code. See [Section 24.66.1.5](#).

- **scope**

Return a fresh Tag whose `stop` will be invoked at the end of the current scope. This function is likely to be removed. See [Section 24.66.1.4](#).

- **stop(*result* = void)**

Stop any code tagged by `this`. If some *result* was specified, let stopped code return *result* as value. See [Section 24.66.1.1](#).

- **tags**

All the undeclared tags are created as slots in this object. Using this feature is discouraged.

```
{
    assert ("brandNewTag" not in Tag.tags.localSlotNames);
    brandNewTag: 1;
    assert ("brandNewTag" in Tag.tags.localSlotNames);
    assert (Tag.tags.brandNewTag.isA(Tag));
};
```

urbiscript
Session

- **unblock**
Unblock `this`. See [Section 24.66.1.2](#).
- **unfreeze**
Unfreeze code tagged by `this`. See [Section 24.66.1.3](#).

24.66.4 Hierarchical tags

Tags can be arranged in a parent/child relationship: any operation done on a tag — freezing, stopping, ... is also performed on its descendants. Another way to see it is that tagging a piece of code with a child will also tag it with the parent. To create a child Tag, simply clone its parent.

urbiscript
Session

```
var parent = Tag.new |
var child = parent.clone |

// Stopping parent also stops children.
{
  parent: {sleep(100ms); echo("parent")},
  child: {sleep(100ms); echo("child")},
  parent.stop;
  sleep(200ms);
  echo("end");
};

[00000001] *** end

// Stopping child has no effect on parent.
{
  parent: {sleep(100ms); echo("parent")},
  child: {sleep(100ms); echo("child")},
  child.stop;
  sleep(200ms);
  echo("end");
};

[00000002] *** parent
[00000003] *** end
```

Hierarchical tags are commonly laid out in slots so as to reflect their tag hierarchy.

urbiscript
Session

```
var a = Tag.new;
var a.b = a.clone;
var a.b.c = a.b.clone;

a:    foo; // Tagged by a
a.b:  bar; // Tagged by a and b
a.b.c: baz; // Tagged by a, b and c
```

24.67 Timeout

Timeout objects can be used as [Tags](#) to execute some code in limited time.

24.67.1 Prototypes

- [Tag](#)

24.67.2 Construction

At construction, a Timeout takes a duration, and a [Boolean](#) stating whether an exception should be thrown on timeout (by default, it does).

urbiscript
Session

```
Timeout.new(300ms);
[00000000] Timeout_0x953c1e0
Timeout.new(300ms, false);
[00000000] Timeout_0x953c1e0
```

24.67.3 Examples

Use it as a tag:

```
var t = Timeout.new(300ms);
[00000000] Timeout_0x133ec0
t: {
    echo("This will be displayed.");
    sleep(500ms);
    echo("This will not.");
};
[00000000] *** This will be displayed.
[00000007:error] !!! Timeout_0x133ec0 has timed out.
```

urbiscript
Session

The same Timeout, t can be reused. It is armed again each time it is used to tag some code.

```
t: { echo("Open"); sleep(1s); echo("Close"); };
[00000007] *** Open
[00000007:error] !!! Timeout_0x133ec0 has timed out.
t: { echo("Open"); sleep(1s); echo("Close"); };
[00000007] *** Open
[00000007:error] !!! Timeout_0x133ec0 has timed out.
```

urbiscript
Session

Even if exceptions have been disabled, you can check whether the count-down expired with `timedOut`.

```
t:sleep(500ms);
[00000007:error] !!! Timeout_0x133ec0 has timed out.
if (t.timedOut)
    echo("The Timeout expired.");
[00000000] *** The Timeout expired.
```

urbiscript
Session

24.67.4 Slots

- `launch`
Fire `this`.

24.68 Traceable

Objects that have a concept of backtrace.

This object, made to serve as prototype, provides a definition of backtrace which can be filtered based on the desired level of verbosity.

This prototype is not made to be constructed.

24.68.1 Slots

- `backtrace`

A call stack as a [List of StackFrames](#). Used by `Exception.backtrace` and `Job.backtrace`.

```
try
{
    [1].map(closure (v) { throw Exception.new("Ouch") })
}
```

urbiscript
Session

```

catch (var e)
{
  for| (var sf: e.backtrace)
    echo(sf.name)
};
[00000001] *** map

```

- **hideSystemFiles**

Remove system files from the backtrace if this value equals `true`. Defaults to `true`.

urbiscript
Session

```

Traceable.hideSystemFiles = false |

try
{
  [1].map(closure (v) { throw Exception.new("Ouch") })
}
catch (var e)
{
  for| (var sf: e.backtrace)
    echo(sf.name)
};
[00000002] *** f
[00000003] *** each|
[00000004] *** map

```

24.69 TrajectoryGenerator

The trajectory generators change the value of a given variable from an *initial value* to a *target value*. They can be *open-loop*, i.e., the intermediate values depend only on the initial and/or target value of the variable; or *closed-loop*, i.e., the intermediate values also depend on the current value value of the variable.

Open-loop trajectories are insensitive to changes made elsewhere to the variable. Closed-loop trajectories are sensitive to changes made elsewhere to the variable — for instance when the human physically changes the position of a robot’s motor.

Trajectory generators are not made to be used directly, rather use the “continuous assignment” syntax (Section 23.11).

24.69.1 Prototypes

- **Object**

24.69.2 Examples

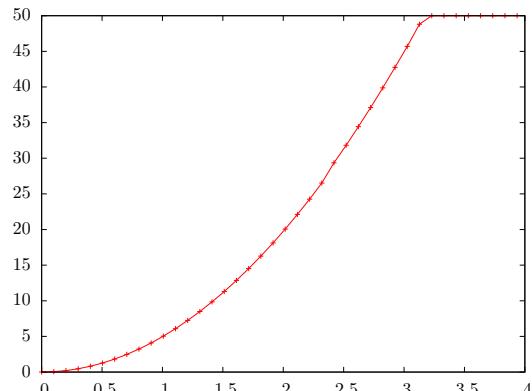
24.69.2.1 Accel

The `Accel` trajectory reaches a target value at a fixed acceleration (`accel` attribute).

```

var y = 0;
y = 50 accel:10,

```

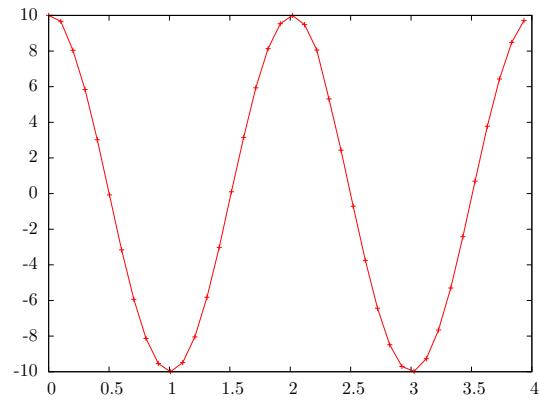


24.69.2.2 Cos

The **Cos** trajectory implements a cosine around the target value, given an amplitude (**ampli** attribute) and period (**cos** attribute).

This trajectory is not “smooth”: the initial value of the variable is not taken into account.

```
var y = 0;
y = 0 cos:2s ampli:10,
```

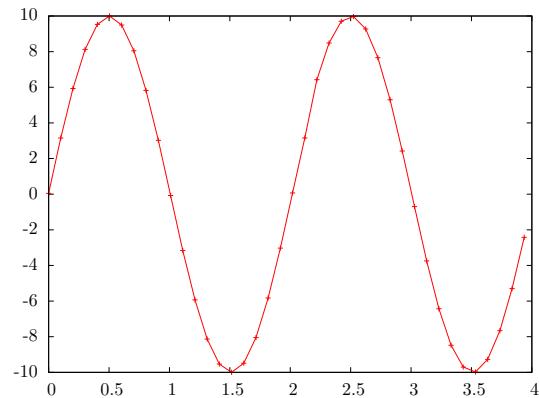


24.69.2.3 Sin

The **Sin** trajectory implements a sine around the target value, given an amplitude (**ampli** attribute) and period (**sin** attribute).

This trajectory is not “smooth”: the initial value of the variable is not taken into account.

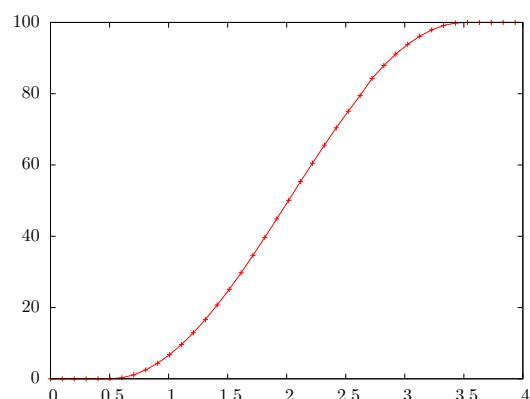
```
var y = 0;
y = 0 sin:2s ampli:10,
```



24.69.2.4 Smooth

The **Smooth** trajectory implements a sigmoid. It changes the variable from its current value to the target value “smoothly” in a given amount of time (**smooth** attribute).

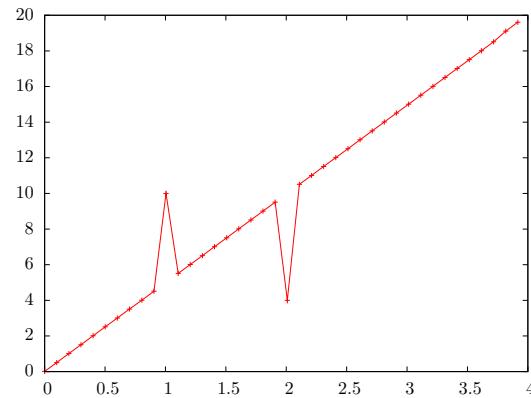
```
var y = 0;
{
  sleep(0.5s);
  y = 100 smooth:3s,
},
```



24.69.2.5 Speed

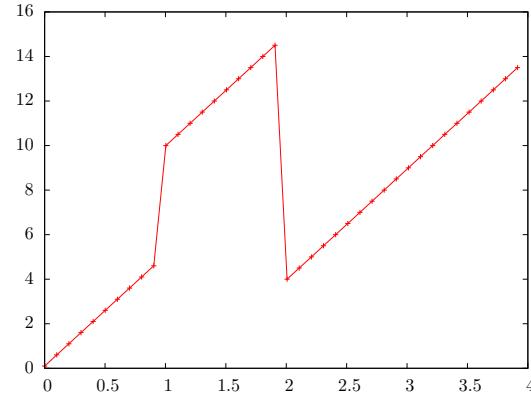
The **Speed** trajectory changes the value of the variable from its current value to the target value at a fixed speed (the **speed** attribute).

```
var y = 0;
assign: y = 20 speed: 5,
{
  sleep(1s);
  y = 10;
  sleep(1s);
  y = 4;
},
```



If the **adaptive** attribute is set to true, then the duration of the trajectory is constantly reevaluated.

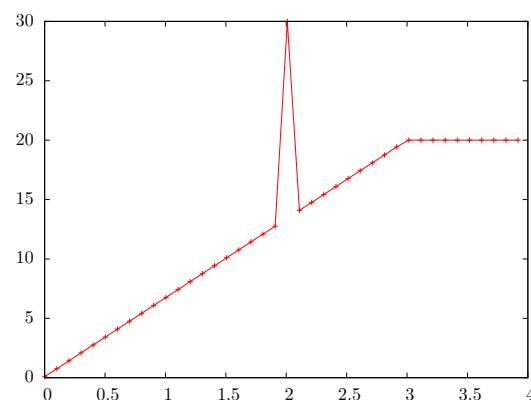
```
var y = 0;
assign: y = 20 speed: 5 adaptive: true,
{
  sleep(1s);
  y = 10;
  sleep(1s);
  y = 4;
},
```



24.69.2.6 Time

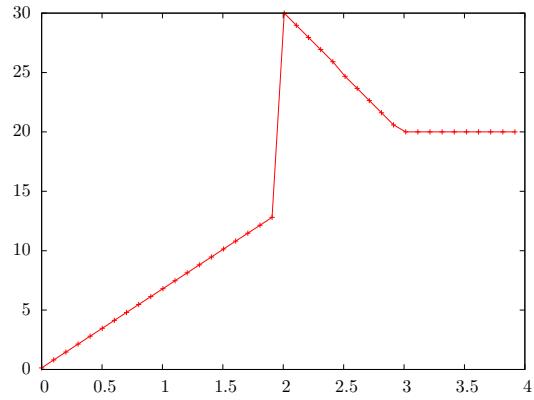
The **Time** trajectory changes the value of the variable from its current value to the target value within a given duration (the **time** attribute).

```
var y = 0;
assign: y = 20 time:3s,
{
  sleep(2s);
  y = 30;
},
```



If the **adaptive** attribute is set to true, then the duration of the trajectory is constantly reevaluated.

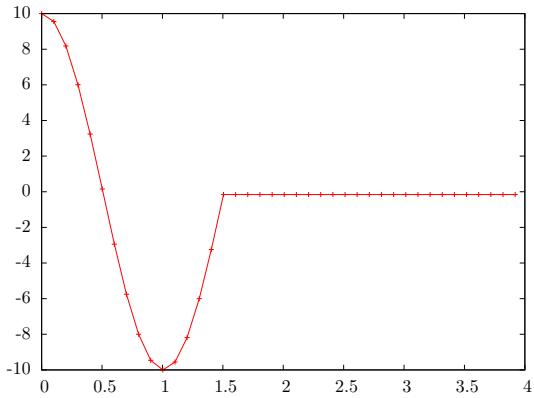
```
var y = 0;
assign: y = 20 time:3s adaptive: true,
{
    sleep(2s);
    y = 30;
},
```



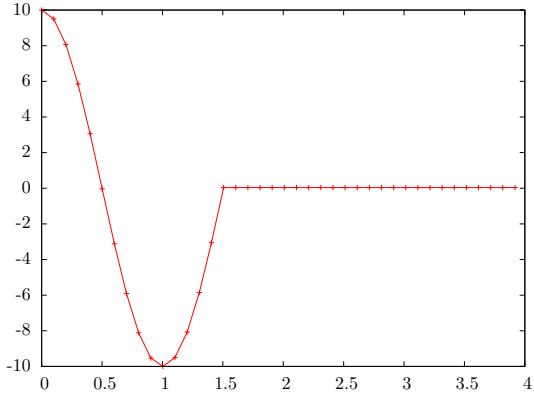
24.69.2.7 Trajectories and Tags

Trajectories can be managed using [Tags](#). Stopping or blocking a tag that manages a trajectory kill the trajectory.

```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
    sleep(1.5s);
    assign.stop;
},
```

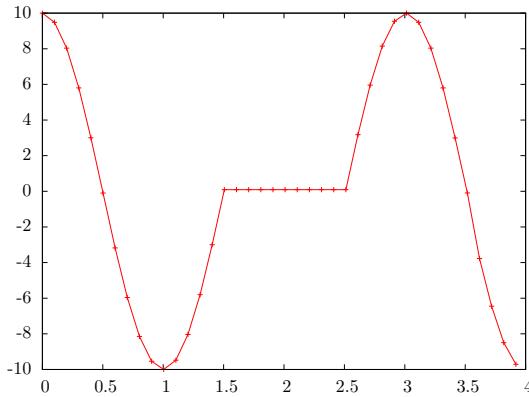


```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
    sleep(1.5s);
    assign.block;
    sleep(1s);
    assign.unblock;
},
```



When a trajectory is frozen, its local time is frozen too, the movement proceeds from where it was rather than from where it would have been had it been not frozen.

```
var y = 0;
assign: y = 0 cos:2s ampli:10,
{
    sleep(1.5s);
    assign.freeze;
    sleep(1s);
    assign.unfreeze;
},
```



24.69.3 Construction

You are not expected to construct trajectory generators by hand, using modifiers is the recommended way to construct trajectories. See [Section 23.11](#) for details about trajectories, and see [Section 24.69.2](#) for an extensive set of examples.

24.69.4 Slots

- **Accel**

This class implements the `Accel` trajectory ([Section 24.69.2.1](#)). It derives from `OpenLoop`.

- **ClosedLoop**

This class factors the implementation of the *closed-loop* trajectories. It derives from `TrajectoryGenerator`.

- **OpenLoop**

This class factors the implementation of the *open-loop* trajectories. It derives from `TrajectoryGenerator`.

- **Sin**

This class implements the `Cos` and `Sin` trajectories ([Section 24.69.2.2](#), [Section 24.69.2.3](#)). It derives from `OpenLoop`.

- **Smooth**

This class implements the `Smooth` trajectory ([Section 24.69.2.4](#)). It derives from `OpenLoop`.

- **SpeedAdaptive**

This class implements the `Speed` trajectory when the `adaptive` attribute is given ([Section 24.69.2.5](#)). It derives from `ClosedLoop`.

- **Time**

This class implements the non-adaptive `Speed` and `Time` trajectories ([Section 24.69.2.5](#), [Section 24.69.2.6](#)). It derives from `OpenLoop`.

- **TimeAdaptive**

This class implements the `Time` trajectory when the `adaptive` attribute is given ([Section 24.69.2.6](#)). It derives from `ClosedLoop`.

24.70 Triplet

A *triplet* (or *triple*) is a container storing three objects.

24.70.1 Prototype

- **Tuple**

24.70.2 Construction

A `Triplet` is constructed with three arguments.

```
Triplet.new(1, 2, 3);
[00000001] (1, 2, 3)

Triplet.new(1, 2);
[00000003:error] !!! new: expected 3 arguments, given 2

Triplet.new(1, 2, 3, 4);
[00000003:error] !!! new: expected 3 arguments, given 4
```

urbiscript
Session

24.70.3 Slots

- `first`

Return the first member of the pair.

```
Triplet.new(1, 2, 3).first == 1;
Triplet[0] === Triplet.first;
```

Assertion
Block

- `second`

Return the second member of the triplet.

```
Triplet.new(1, 2, 3).second == 2;
Triplet[1] === Triplet.second;
```

Assertion
Block

- `third`

Return the third member of the triplet.

```
Triplet.new(1, 2, 3).third == 3;
Triplet[2] === Triplet.third;
```

Assertion
Block

24.71 Tuple

A `tuple` is a container storing a fixed number of objects. Examples include `Pair` and `Triplet`.

24.71.1 Prototype

- `Object`

24.71.2 Construction

The `Tuple` object is not meant to be instantiated, its main purpose is to share code for its descendants, such as `Pair`. Yet it accepts its members as a list.

```
var t = Tuple.new([1, 2, 3]);
[00000000] (1, 2, 3)
```

urbiscript
Session

The output generated for a `Tuple` can also be used to create a `Tuple`. Expressions are put inside parentheses and separated by commas. One extra comma is allowed after the last element. To avoid confusion between a 1 member `Tuple` and a parenthesized expression, the extra comma must be added. `Tuple` with no expressions are also accepted.

```
// not a Tuple
(1);
[00000000] 1

// Tuples
```

urbiscript
Session

```

();
[00000000] ()
(1,);
[00000000] (1,)
(1, 2);
[00000000] (1, 2)
(1, 2, 3, 4,);
[00000000] (1, 2, 3, 4)

```

24.71.3 Slots

- **asString**

The string ‘(*first*, *second*, …, *last*)’, using `asPrintable` to convert members to strings.

Assertion Block

```

().asString == "()";
(1,).asString == "(1,)";
(1, 2).asString == "(1, 2)";
(1, 2, 3, 4,).asString == "(1, 2, 3, 4)";

```

- **matchAgainst(handler, pattern)**

Pattern matching on members. See [Pattern](#).

urbiscript Session

```

{
    // Match a tuple.
    (var first, var second) = (1, 2);
    assert { first == 1; second == 2 };
}

```

- **size**

Number of members.

Assertion Block

```

().size == 0;
(1,).size == 1;
(1, 2, 3, 4).size == 4;
(1, 2, 3, 4,).size == 4;

```

- **[]’(index)**

Return the *index*-th element. *index* must be in bounds.

Assertion Block

```

(1, 2, 3)[0] == 1;
(1, 2, 3)[1] == 2;

```

- **[]=’(index, value)**

Set (and return) the *index*-th element to *value*. *index* must be in bounds.

urbiscript Session

```

{
    var t = (1, 2, 3);
    assert
    {
        (t[0] = 2) == 2;
        t == (2, 2, 3);
    };
}

```

- **<’(other)**

Lexicographic comparison between two tuples.

Assertion Block

```

(0, 0) < (0, 1);
(0, 0) < (1, 0);
(0, 1) < (1, 0);

```

- `'==' (other)`

Whether `this` and `other` have the same contents (equality-wise).

```
(1, 2) == (1, 2);
!((1, 1) == (2, 2));
```

Assertion Block

- `'*' (value)`

Return a Tuple in which all elements of `this` are multiplied by a `value`.

```
(0, 1, 2, 3) * 3 == (0, 3, 6, 9);
(1, "foo") * 2 == (2, "foofoo");
```

Assertion Block

- `'+' (other)`

Return a Tuple in which each element of `this` is summed with its corresponding element in the `other` Tuple.

```
(0, 1) + (2, 3) == (2, 4);
(1, "foo") + (2, "bar") == (3, "foobar");
```

Assertion Block

24.72 UObject

UObject is used by the UObject API (see Part I) to represent a bound C++ instance.

All the UObject are copied under a unique name as slots of `Global.uobjects`.

24.72.1 Prototypes

- `Object`

24.72.2 Slots

- `uobjectName` Unique name assigned to this object. This is also the slot name of `Global.uobjects` containing this UObject.

- `searchPath`

The search-path for UObject files (see Section 22.1) as a `List` of `Paths`. See also `System.loadLibrary` and `System.loadModule`.

```
UObject.searchPath.isA(List);
UObject.searchPath[0].isA(Path);
```

Assertion Block

24.73 uobjects

This object serves only to store the UObject that are bound into the system (plug or remote).

24.73.1 Prototypes

- `Object`

24.73.2 Slots

- `asuobjects`
Return `this`.

- `clearStats`
Reset counters.

- `connectionStats`

- `enableStats(bool)`
Depending on `bool`, enable or disable the statistics gathering.
- `getStats`
Return a dictionary of all bound C++ functions called, including timer callbacks, along with the average, min, max call durations, and the number of calls.
- `resetConnectionStats`

- `searchPath`

- `setTrace(bool)`

24.74 UValue

The UValue object is used internally by the UObject API and is mostly hidden from the user.

24.75 UVar

This class is used internally by the UObject middleware (see [Part I](#)) to represent a variable that can be hooked in C++. Each slot on which a C++ `urbi::UVar` exists contains an instance of this class.

Instances of `UVar` are mostly transparent, they appear as the value they contain. Thus, since the `UVar` evaluates to the contained value, you must use `getSlot` to manipulate the `UVar` itself.

24.75.1 Construction

To instantiate a new `UVar`, pass the owner object and the slot name to the constructor.

urbiscript
Session

```
UVar.new(Global, "x") |
Global.x = 5;
[000000001] 5
x;
[000000002] 5
```

24.75.2 Prototypes

- [Object](#)

24.75.3 Slots

- `copy(targetObject, targetName)`

Copy the `UVar` to the slot `targetName` of object `targetObject`. Since the `UVar` is using properties, you must use this method to copy it to an other location.

- `hookChanged`

A Boolean that states whether future `UVar` instantiations support the `changed` event.
Code like:

urbiscript
Session

```
at (headTouch.val->changed? if headTouch.val)
tts.say("ouch");
```

works, but costs one `at` per UVar (whose cost is light in recent version of Urbi SDK). Set `hookChanged` to false *before* instantiating new UVars to disable this feature; you may also set it to true to re-enable the feature for UVars that will be instantiating afterward. See also the environment variable `URBI_UVAR_HOOK_CHANGED`, [Section 22.1.2](#).

- `notifyAccess(onAccess)`

Similar to the C++ `UNotifyAccess`, calls `onAccess` each time the UVar is accessed (read).

```
var Global.counter = 0|
UVar.new(Global, "access")|
var accessHandle = &access.notifyAccess(closure() {
    Global.access = Global.counter += 1
})|
assert
{
    access == 1;
    access == 2;
    access == 3;
};
&access.removeNotifyAccess(accessHandle)|;
assert
{
    access == 3;
    access == 3;
};
```

urbiscript
Session

- `notifyChange(onChange)`

Similar to the C++ `UNotifyChange` for a non-owned UVar (see [Section 5.5](#)), register `onChange` and call it each time this UVar is written to. Return an identifier that can be passed to `removeNotifyChange` to unregister the callback.

```
UVar.new(Global, "y")|
var handle = Global.&y.notifyChange(closure() {
    echo("The value is now " + Global.y)
})|
Global.y = 12;
[00000001] *** The value is now 12
[00000002] 12
Global.&y.removeNotifyChange(handle)|;
Global.y = 13;
[00000003] 13
```

urbiscript
Session

- `notifyChangeOwned(onChangeOwned)`

Similar to the C++ `UNotifyChange` for an owned UVar (see [Section 5.5](#)), register `onChange` and call it each time this UVar is written to. Return an identifier that can be passed to `removeNotifyChange` to unregister the callback.

- `removeNotifyAccess(id)`

Disable the notification installed as `id` by `notifyAccess`.

- `removeNotifyChange(id)`

Disable the notification installed as `id` by `notifyChange`.

- `removeNotifyChangeOwned(id)`

Disable the notification installed as `id` by `notifyChangeOwned`.

- `owned`

True if the UVar is in *owned mode*, that is if it contains both a sensor and a command value.

24.76 void

The special entity `void` is an object used to denote “no value”. It has no prototype and cannot be used as a value. In contrast with `nil`, which is a valid object, `void` denotes a value one is not allowed to read.

24.76.1 Prototypes

None.

24.76.2 Construction

`void` is the value returned by constructs that return no value.

Assertion Block

```
void.isVoid;
{} .isVoid;
{if (false) 123}.isVoid;
```

24.76.3 Slots

- `acceptVoid`

Trick `this` so that, even if it is `void` it can be used as a value. See also `unacceptVoid`.

urbiscript Session

```
void.foo;
[00096374:error] !!! unexpected void
void.acceptVoid.foo;
[00102358:error] !!! lookup failed: foo
```

- `isVoid`

Whether `this` is `void`. Therefore, return `true`.

Assertion Block

```
void.isVoid;
void.acceptVoid.isVoid;
! 123.isVoid;
```

- `unacceptVoid`

Remove the magic from `this` that allowed to manipulate it as a value, even if it is `void`. See also `acceptVoid`.

urbiscript Session

```
void.acceptVoid.unacceptVoid.foo;
[00096374:error] !!! unexpected void
```

Chapter 25

Communication with ROS

This chapter is not an introduction to using ROS from Urbi, see [Chapter 16](#) for a tutorial.

Urbi provides a set of tools to communicate with ROS (Robot Operating System). For more information about ROS, please refer to <http://www.ros.org>. Urbi, acting as a ROS node, is able to interact with the ROS world.

Requirements You need to have installed ROS (possibly a recent version), and compiled all of the common ROS tools (`rxconsole`, `roscore`, `roscpp`, ...).

You also need to have a few environment variables set, normally provided with ROS installation: `ROS_ROOT`, `ROS_MASTER_URI` and `ROS_PACKAGE_PATH`.

Usage The classes are implemented as UObjects (see [Part I](#)): `Ros`, `Ros.Topic`, and `Ros.Service`.

This module is loaded automatically if `ROS_ROOT` is set in your environment. If `roscore` is not launched, you will be warned and Urbi will check regularly for `roscore`.

If for any reason you need to load this module manually, use:

```
loadModule("urbi/ros");
```

urbiscript
Session

25.1 Ros

This object provides some handy tools to know the status of `roscore`, to list the different nodes, topics, services, ... It also serves as a namespace entry point for ROS entities, such as `Ros.Topic` and so forth.

25.1.1 Construction

There is no construction, since this class only provides a set of tools related to ROS in general, or the current node (which is unique per instance of Urbi).

25.1.2 Slots

- `checkMaster`

Whether `roscore` is accessible.

- `name`

The name of the current node associated with Urbi (as a string). The name of an Urbi node is generally composed of '`/urbi+random`' sequence.

- `nodes`

A [Dictionary](#) containing all the nodes known by `roscore`. Each key is a node name. Its associated value is another [Dictionary](#), with the following keys: `publish`, `subscribe`, `services`. Each of these keys is associated with a list of topics or services.

Assertion
Block

```
"/rosout" in Ros.nodes;
Ros.name in Ros.nodes;
```

- Service

The `Ros.Service` class.

- services

A `Dictionary` containing all the services known by `roscore`. Each key is the name of a service, and the associated value is a list of nodes that provide this service.

urbiscript
Session

```
var services = Ros.services|
var name = Ros.name|;
assert
{
    "/rosout/get_loggers" in services;
    "/rosout/set_logger_level" in services;
    (name + "/get_loggers") in services;
    (name + "/set_logger_level") in services;
};
```

- Topic

The `Ros.Topic` class.

- topics

A `Dictionary` containing all the topics advertised to `roscore`. Each key is the name of a topic, and the associated value is another `Dictionary`, with the following keys: `subscribers`, `publishers`, `type`.

urbiscript
Session

```
var topics = Ros.topics|;
topics.keys;
[03316144] ["/rosout_agg"]

// The actual value of the "type" field depends on the ROS version
// (the location changed, but the type is the same):
// - "roslib/Log" for CTurtle.
// - "rosgraph_msgs/Log" for Diamondback and later.
topics["/rosout_agg"];
[03325634] ["publishers" => ["/rosout"], \
"subscribers" => [], \
"type" => "rosgraph_msgs/Log"]
```

25.2 Ros.Topic

This `UObject` provides a handy way to communicate through ROS topics, by subscribing to existent topics or advertising to them.

25.2.1 Construction

To create a new topic, call `Ros.topic.new` with a string (the name of the topic you want to subscribe to / advertise on).

The topic name can only contain alphanumerical characters, ‘/’ and ‘_’, and cannot be empty. If the topic name is invalid, an exception is thrown and the topic is not created.

Until you decide what you want to do with your topic (`subscribe` or `advertise`), you are free to call `init` to change its name.

25.2.2 Slots

Some slots on this UObject have no interest once the type of instance is determined. For example, you cannot call `unsubscribe` if you `advertise`, and in the same way you cannot call `publish` if you subscribed to a topic.

25.2.2.1 Common

- `name`

The name of the topic provided to `init`.

```
Ros.Topic.new("/test").name == "/test";
```

Assertion Block

- `structure`

Once `advertise` or `subscribe` has been called, this slot contains a template of the message type, with default values for each type (empty strings, zeros, ...). This template is a [Dictionary](#).

```
var logTopic = Ros.Topic.new("/rosout_agg")|;
logTopic.subscribe|;
assert
{
    logTopic.structure.keys
    == ["file", "function", "header", "level", "line", "msg", "name", "topics"];
};
```

Subscription Session

- `subscriberCount`

The number of subscribers for the topic given in `init`; 0 if the topic is not registered.

25.2.2.2 Subscription

- `onMessage`

Event triggered when a new message is received from a subscribed channel, the payload of this event contains the message.

```
var t = Ros.Topic.new("/test")|;
at (t.onMessage?(var m))
    echo(m);
t.subscribe;
```

Subscription Session

- `subscribe`

Subscribe to the provided topic. Throw an exception if it doesn't exist, or if the type advertised by ROS for this topic does not exist. From the call of this method, a direct connection is made between the advertiser and the subscriber which starts deserializing the received messages.

- `unsubscribe`

Unsubscribe from a previously subscribed channel, and set the state of the instance as if `init` was called but not `subscribe`.

25.2.2.3 Advertising

- '`<<`' (*message*)

An alias for `publish`.

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub << ["data" => "Hello world!"];
```

Subscription Session

- **advertise(*type*)**

Tells ROS that this node will advertise on the topic given in `init`, with the type *type*. *type* must be a valid ROS type, such as ‘`roslib/Log`’. If *type* is an empty string, the method will try to check whether a node is already advertising on this topic, and its type.

urbiscript Session

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub.structure;
[00670809] ["data" => ""]
```

- **onConnect(*name*)**

Event triggered when the current instance is advertising on a topic and a node subscribes to this topic. The payload *name* is the name of the node that subscribed. See also [onDisconnect](#).

urbiscript Session

```
var p = Ros.Topic.new("/test/publisher")|;
at sync (p.onConnect?(var n))
  echo("%s has subscribed to %s" % [n, p.name]);
// The structure is not defined, yet.
assert (p.structure.isVoid);
p.advertise("std_msgs/String");
// The structure is ready to be used.
assert (p.structure == ["data" => ""]);

var s = Ros.Topic.new("/test/publisher")|;
```

Subscription and unsubscription are asynchronous. To make them synchronous, we [waituntil](#) the connection event is sent. The first idea:

urbiscript Session

```
s.subscribe;
waituntil (p.onConnect?);
```

is wrong: the event (fired in the first line) might be accomplished before the second line is even started. In that case, Urbi will remain suspended, waiting for an already passed event. Rather, we will “[waituntil](#)” in background *before* subscribing, and rely on the fact that scopes “[wait](#)” for all the background statements to be finished ([Section 23.1.7.2](#)):

urbiscript Session

```
{
  waituntil (p.onConnect?),
  s.subscribe;
};
```

[00077308] *** /urbi_1317980847216045000 has subscribed to /test/publisher

- **onDisconnect(*name*)**

Event triggered when the current instance is advertising on a topic, and a node unsubscribes from this topic. The payload *name* is the name of the node that unsubscribed. See also [onConnect](#). The following example is a continuation of the previous one.

urbiscript Session

```
at (p.onDisconnect?(var n))
  echo("%s has unsubscribed to %s" % [n, p.name]);

{
  waituntil (p.onDisconnect?),
  s.unsubscribe;
};
```

[00077308] *** /urbi_1317980847216045000 has unsubscribed to /test/publisher

- **publish(*message*)**

Publish *message* to the current topic. This method is usable only if [advertise](#) was called.

The message must follow the same structure as the one in the slot `structure`, or it will throw an exception telling which key is missing / wrong in the dictionary.

```
var stringPub = Ros.Topic.new("/mytest")|;
stringPub.advertise("std_msgs/String");
stringPub.publish(["data" => "Hello world!"]);
```

urbiscript
Session

- `unadvertise`

Tells ROS that we stop publishing on this topic. The `Object` then gets back to a state where it could call `init`, `subscribe` or `advertise`.

25.2.3 Example

This is a typical example of the creation of a publisher, a subscriber, and message transmission between both of them.

First we need to declare our Publisher, the topic name is '`/example`', and the type of message that will be sent on this topic is '`std_msgs/String`'. This type contains a single field called '`data`', holding a string. We also set up handlers for `onConnect` and `onDisconnect` to be noticed when someone subscribes to us.

```
var publisher = Ros.Topic.new("/example")|
at (publisher.onConnect?(var name))
echo(name[0,5] + " is now listening on " + publisher.name);
at (publisher.onDisconnect?(var name))
echo(name[0,5] + " is no longer listening on " + publisher.name);
publisher.advertise("std_msgs/String");
```

urbiscript
Session

Then we subscribe to the freshly created topic, and for each message, we display the '`data`' section (which is the content of the message). Thanks to the previous `at` above, a message is displayed at subscription time.

```
var subscriber = Ros.Topic.new("/example")|
at (subscriber.onMessage?(var m))
echo(m["data"]);
subscriber.subscribe;
// Let the "is now listening" message arrive.
sleep(200ms);
[00026580] *** /urbi is now listening on /example
```

urbiscript
Session

The structure for the messages are, of course, equal between the subscriber and the publisher.

```
subscriber.structure == publisher.structure;
```

Assertion
Block

Now we can send a message, and get it back through the `at` in the section above. To do this we first copy the template structure and then fill the field '`data`' with our message.

```
var message = publisher.structure.new;
[00098963] ["data" => ""]
message["data"] = "Hello world!|;

// publish the message.
publisher << message;
// Leave some time to asynchronous communications before shutting down.
sleep(200ms);
[00098964] *** Hello world!

subscriber.unsubscribe;
// Let the "is no longer" message arrive.
sleep(200ms);
[00252566] *** /urbi is no longer listening on /example
```

urbiscript
Session

25.3 Ros.Service

This `UObject` provides a handy way to call services provided by other ROS nodes.

25.3.1 Construction

To create a new instance of this object, call `Ros.Service.new` with a string representing which service you want to use, and a Boolean stating whether the connection between you and the service provider should be kept opened (pass `true` for better performances on multiple requests).

The service name can only contain alphanumerical characters, ‘/’, ‘_’, and cannot be empty. If the service name is invalid, an exception is thrown, and the object is not created.

Then if the service does not exist, an other exception is thrown. Since the initialization is asynchronous internally, you need to wait for `service.initialized` to be true to be able to call `request`.

25.3.2 Slots

- `initialized`

Boolean. Whether the method `request` is ready to be called, and whether `resStruct` and `reqStruct` are filled.

urbiscript
Session

```
var logService = Ros.Service.new("/rosout/get_loggers", false)|;
waituntil(logService.initialized);
```

- `name`

The name of the current service.

urbiscript
Session

```
logService.name;
[00036689] "/rosout/get_loggers"
```

- `reqStruct`

Get the template of the request message type, with default values for each type (empty strings, zeros, …). This template is made of dictionaries.

urbiscript
Session

```
logService.reqStruct;
[00029399] [ => ]
```

- `request(req)`

Synchronous call to the service, providing `req` as your request (following the structure of `reqStruct`). The returned value is a `Dictionary` following the structure of `resStruct`, containing the response to your request.

urbiscript
Session

```
for (var item in logService.request([=>])["loggers"])
    echo(item);
[00034567] *** ["level" => "INFO", "name" => "ros"]
[00034571] *** ["level" => "INFO", "name" => "ros.roscpp"]
[00034575] *** ["level" => "INFO", "name" => "ros.roscpp.roscpp_internal"]
[00034586] *** ["level" => "WARN", "name" => "ros.roscpp.superdebug"]
```

- `resStruct`

Get the template of the response message type, with default values for each type (empty strings, zeros, …). This template is made of dictionaries.

urbiscript
Session

```
logService.resStruct;
[00029399] ["loggers" => []]
```

Chapter 26

Gostai Standard Robotics API

This section aims at clarifying the naming conventions in Urbi Engines for standard hardware/-software devices and components implemented as UObject and the corresponding methods/attributes/events to access them. The list of available hardware types and software component is increasing and this document will be updated accordingly. Please contact us directly, should you be working on a component not described or closely related to one described here:

standard@gostai.com

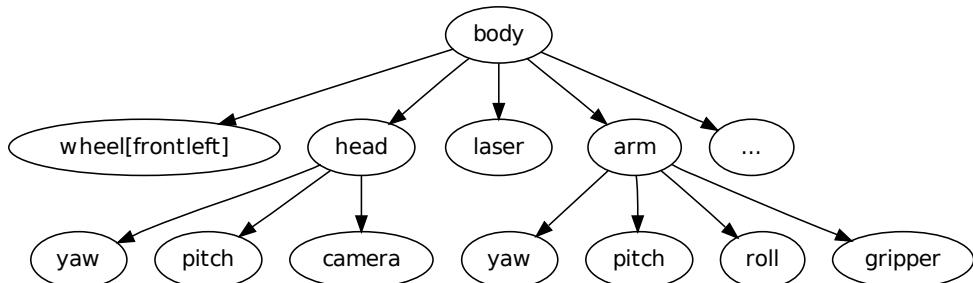
Any implementation of an Urbi server must comply with the latest version of this standard to get the “Urbi Ready” certification from Gostai S.A.S.

Gostai S.A.S. is currently the only authority which has the ability to deliver an “Urbi Ready” certification.

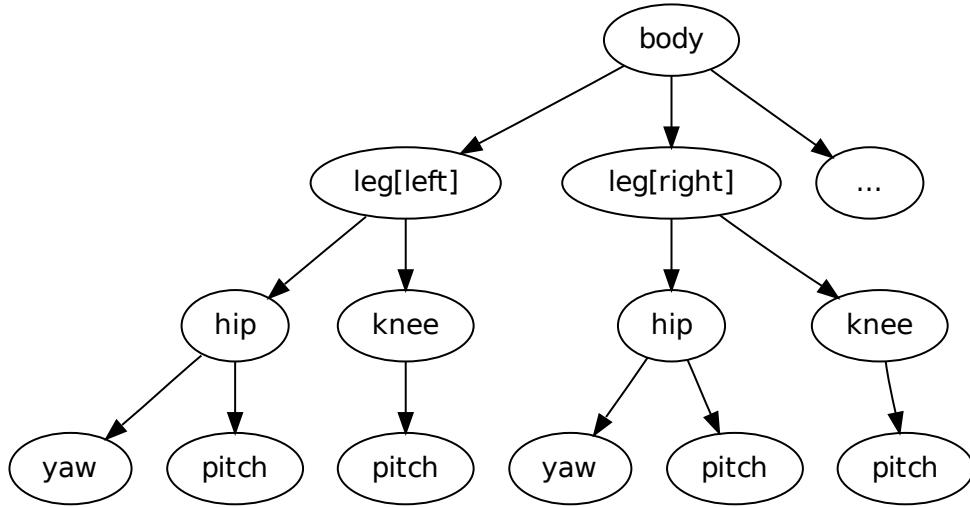
“Urbi Ready” and the associated logo are trademarks of Gostai S.A.S. and should not be used or displayed in any way without an explicit written agreement from Gostai.

26.1 The Structure Tree

The robot will be described as a set of *components* organized in a hierarchical structure called the *structure tree*. The relationship between a component and a sub-component in the tree is a ‘part-of’ inclusion relationship. From the point of view of Urbi, each component in the tree is an object, and it contains attributes pointing to its sub-components. Here is an example illustrating a part of a hierarchy that could be found with a wheeled robot with a gripper:



And here is another example for an humanoid robot:



The leaves of the tree are called *devices*, and they usually match physical devices in the robot: motors, sensors, lights, camera, etc. Inside Urbi, the various objects corresponding to the tree components are accessed by following the path of objects inclusions, like in the example below (shortcuts will be described later):

urbiscript
Session

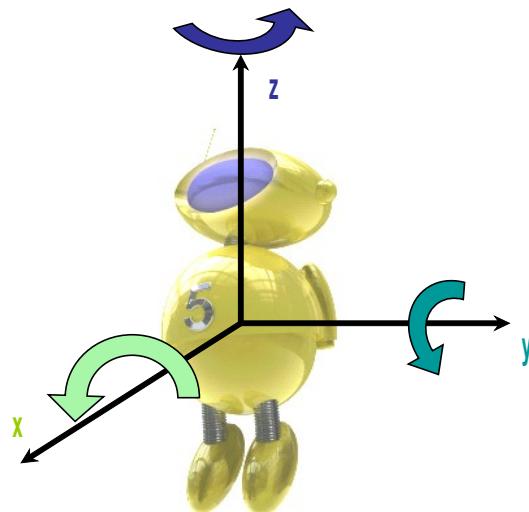
```
body.leg[right].hip.tilt;
body.arm.grip;
body.laser;
// ...
```

The structure tree should not be mistaken for a representation of the kinematics chain of the robot. The kinematics chain is built from a subset of the devices corresponding to motor devices, and it represents spatial connections between them. Except for these motor devices, the structure tree components do not have a direct counterpart in the kinematics chain, or, if they do, it is as a subset of the kinematics chain (for example, `leg[right]` is a subset of the whole kinematics chain).

The goal of this standard is to provide guidelines on how to define the components and the structure tree, knowing the kinematics chain of the robot.

26.2 Frame of Reference

In many cases, it will be necessary to refer to an absolute frame of reference attached to the robot body. To avoid ambiguities, the standard frame of reference will have the following definition:



Origin the center of mass of the robot

X axis oriented towards the front of the robot. If there is a camera, the front is defined by the default direction of the camera, otherwise the front will be seen as the natural frontal orientation for a mobile robot (the direction of “forward” movement). If the robot is not naturally oriented, the X axis will be chosen to match the main axis of symmetry of the robot body and it will be oriented towards the smallest side, typically the top of a cone for example. In case of a perfectly symmetrical body, the X axis can be chosen arbitrarily but a clear mark should be made visible on the robot body to indicate it.

Z axis oriented in the opposite direction from the gravity. If there is no gravity or natural up/down orientation in the environment or normal operation mode of the robot, the Z axis should be chosen in the direction of the main axis of symmetry in the orthogonal plane defined by the X axis, oriented towards the smallest side. In case of a perfectly symmetrical plane, the Z axis can be chosen arbitrarily but a clear mark should be made visible on the robot body to indicate it.

Y axis oriented to make a right-handed coordinate system.

The axes are oriented in a counter-clockwise direction, as depicted in the illustration above.

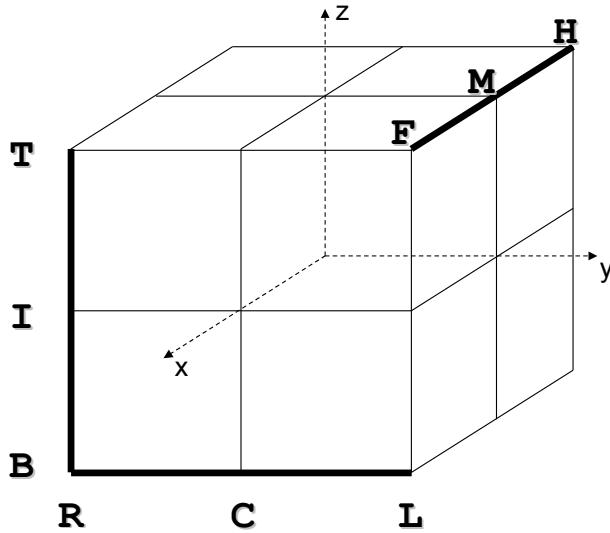
26.3 Component naming

Each component A, which is a sub-component of component B has a name, distinct from the name of all the other components at the same level. This name is a generic designation of what A represents, such as “leg”, “head”, or “finger”.

Using the correct name for each component is a critical part of this standard. No formal rule can be given to find this name for any possible robot configuration. However, this document includes a table covering many different possible cases. We recommend that robot manufacturers pick from this table the name that fits the most the description of their component.

26.4 Localization

When two identical components A1 and A2, such as the two legs of an humanoid robots, are present in the same sub-component B, an extra node is inserted in the hierarchy to differentiate them. This node is of the [Localizer](#) type, and provides a `[]` operator, taking a [Localization](#) argument, used to access each of the identical sub-components. The Urbi SDK provides an implementation for the [Localizer](#) and [Localization](#) classes. When possible, localization should be simple geometrical qualifier like *right/center/left*, *front/middle/back* or *up/in-between/down*. Note that “right” or “front” are understood here from the point of view of a man standing and looking in the direction of the X-axis of the robot, and *up/pown* matches the Z-axis, as depicted in the figure below:

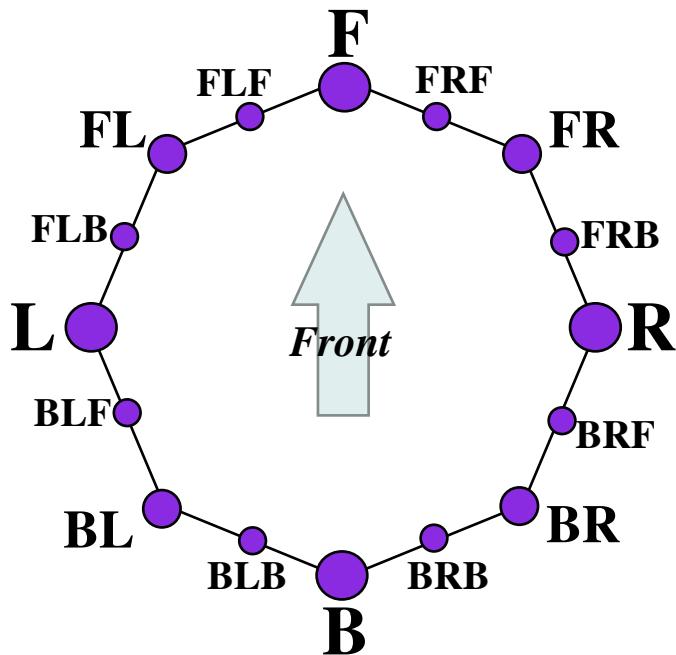


Several geometric qualifiers can be used at the same time to further refine the position. In this case, multiple Localizer nodes are used. As a convention, height information (U/I/D) comes first, followed by depth information (F/M/H), and then side information (R/C/L).

urbiscript
Session

```
// Front-left wheel of a four-wheeled robot:  
robot.body.wheel[front][left];  
// Front laser of a robot equipped with multiple sonars:  
robot.body.laser[front];  
// Left camera from a robot with a stereo camera at the end of an arm:  
robot.body.arm.camera[left];  
// Top-left LED of the left eye.  
robot.body.head.eye[left].led[up][left].val = 1;  
// Touch sensor at the end of the front-left leg of a four-legged robot:  
robot.body.leg[front][left].foot.touch;
```

You can further qualify a side+depth localization with an additional F/B side information. This can be used in the typical layout below:



This dual positioning using side+depth can also be used to combine side+height or height+depth information.

Layouts with a sequence of three or more identical components can use numbers as their Localization, starting from 0. The smaller the number, the closer to the front, up, or left. For instance, an insectoid robot with 3 legs on each side will use `robot.body.leg[left][0]` to address the frontleft leg.

Layouts with identical components arranged in a circle can also use numeric localization. The component with index 0 should be the uppermost, or front-most if non applicable. Index should increase counterclockwise.

Some components like spines or tails are highly articulated with a set of identical sub-components. When talking about these sub-components, the above localization should be replaced by an array with a numbering starting at 0. The smaller the number, the closer the sub-component is to the robot main body. For surface-like sub-components, like skin touch sensors, the array can be two dimensional.

Other possible localization for sensors are the X, Y and Z axis themselves, like for example for an accelerometer or a gyro sensor, available in each of the three directions.

```
robot.body.accel[x]; // accelerometer in the x direction
```

urbiscript
Session

Examples of component names including localization:

```
leg[right], leg[left];
finger[right], finger[center], finger[left]; // three-finger hand
joint[0], joint[1] ... joint[5] // from tail
touch[478][124] // from skin
accel[x], accel[y], accel[z]; // typical accelerometer
gyro[x], gyro[y], gyro[z]; // typical gyro sensor
```

urbiscript
Session

26.5 Interface

An *interface* describes some aspects of a type of device, by specifying the slots and methods that implementations must provide. Each child node of the component hierarchy should implement at least one interface.

For example, for a joint, we can have a “Swivel” interface, used to define patella joints. For the robot body itself, we have a [Interface.Mobile](#) interface describing mobile robots, which includes some standard way of requesting a move forward, a turn, etc.

In short, interfaces are standard Urbi objects that components can inherit from to declare that they have some functionalities.

The following pages describe a few of the most standard interfaces. Each device in the component hierarchy which falls within the category of an interface should implement it.

Each interface defines slots, which can be functions, events or plain data. Some of those slots are optional.

26.5.1 AudioIn

The AudioIn interface groups every information relative to microphones.

- **duration**

Amount of sound in the val attribute, expressed in *ms*.

- **gain**

(Optional.) Microphone gain amplification (expressed between 0 and 1).

- **val**

Binary value corresponding to the sound heard, expressed in the current unit (wav, mp3...). The unit can be changed like any other regular unit in Urbi.

The content is the sound heard by the microphone since the last update event.

26.5.2 AudioOut

The AudioOut interface groups every information relative to speakers.

- **playing**

This is a Boolean value which is true when there is a sound currently playing (the buffer is not empty)

- **remain**

The amount of time remaining to play in the speaker sound buffer (expressed in *ms* as a default unit).

- **val**

The speaker value, expressed as a binary, in the format given by the binary header during the assignment.

Speakers are write-only devices, so there is not much sense in reading the content of this attribute. At best, it returns the remaining sound to be played if it is not over yet, but this is not a requirement.

- **volume**

(Optional.) Volume of the play back, in decibels.

26.5.3 Battery

Power source of any kind.

- **capacity**

Storage capacity in Amp.Hour.

- **current**

Current current consumption in Amp.

- **remain**

Estimation of the remaining energy between 0 and 1.

- **voltage**

Current voltage in Volt.

26.5.4 BlobDetector

Ball detectors, marker detectors and various feature-based detectors should all share a similar interface. They extract a part of the image that fits some criteria and define a *blob* accordingly. Here are the typical slots expected:

- **elongation**

(Optional.) Ratio between the main and the second diameter of the blob enveloping ellipse.

- **orientation**

(Optional.) Angle of the main ellipsoid axis of the blob (0 = horizontal), expressed in radians.

- **ratio**

The size of the blob expressed as a normalized image size: 1 = full image, 0 = nothing.

- **threshold**

The minimum value of ratio to decide that the blob is visible.

- **visible**

A Boolean expressing whether there is a blob in the image or not (see [threshold](#)).

- **x**

The x position of the center of the blob in the image

- **y**

The y position of the center of the blob in the image

26.5.5 Identity

Information about the robot identity.

- **kind**

This describes the robot category among: humanoid, four-legged, wheeled, industrial arm. It gives a general idea of the robot family, but does not replace a more systematic probe of available services by investigating the list of attributes of the object.

- **model**

Model of the robot.

- **name**

Name of the robot.

- **serial**

Serial number (if available).

26.5.6 Led

Simple uni-color Led.

- **val**

Led intensity between 0 and 1.

26.5.6.1 RGBLed

Subclass of [Interface.Led](#). Tri-color led.

- **b**

Intensity of the blue component, between 0 and 1.

- **g**

Intensity of the green component, between 0 and 1.

- **r**

Intensity of the red component, between 0 and 1.

26.5.7 Mobile

Mobile robots all share this generic interface to provide high order level motion control capabilities.

26.5.7.1 Blocking API

The following set of functions will block until associated movement is finished. If a function of this set is called while an other is already running, the first call will be canceled: the movement will be interrupted and the call will return.

Many of the functions below take a position argument. This position is expressed as a subset of the six values x, y, z, rho, theta, phi.

- **go(*x*)**

Move approximately *x* meters forward (along the X axis) if *x* is positive, backward otherwise.

- **goTo(*position*)**

Try to reach the given coordinates in the robot frame of reference.

- **goToAbsolute(*position*)**

Try to reach given coordinates in an absolute frame of reference. This frame of reference is maintained using the robot odometry, or any other localization system available.

- **goToChargingStation**

Try to reach the charging station. If the feature is not available, return immediately.

- **position**

The current robot position in the absolute frame of reference.

- **setAbsolutePosition(*position*)**

Force absolute position to the given value.

- **stop**

Stop current movement.

- **turn(*theta*)**

Turn left (in the direction going from the positive X axis to the positive Y axis) approximately *theta* radians. *theta* can be a positive or negative value.

26.5.7.2 Speed-control API

In addition to the previous functions, one can directly set the target movement speed in all linear/angular directions using the 6 scalar variables `xSpeed`, `ySpeed`, `zSpeed`, `yawSpeed`, `pitchSpeed`, `rollSpeed`, or the variable `speed` which contains a list. Writing to one of those slots will abort any function in the blocking API.

Implementations should provide default reasonable values for speed in the slots `defaultXSpeed`, `defaultYSpeed`, `defaultZSpeed`, `defaultYawSpeed`, `defaultPitchSpeed` and `defaultRollSpeed`.

26.5.7.3 Safety

- **watchdog**

Writing any value to `watchdog` will reset the watchdog timer.

- **watchdogInterval**

If non-zero, duration after which the robot will stop if no order was received. Once activated, one has to write to `watchdog`, to one of the speed variables or call one of the blocking API functions every `watchdogInterval` at most, or the `stop` function will be called.

26.5.7.4 State

- **aborted**

(Optional.) Emitted each time a function in the blocking API is aborted because it was preempted by an other movement.

- **finished**

(Optional.) Emitted each time a function in the blocking API finished normally.

- **moving**

True if the robot is currently moving for any reason, false otherwise.

- **unreachable**

(Optional.) Emitted when a function in the blocking API is aborted because the target cannot be reached.

26.5.8 Motor

This interface is used to describe a generic motor controller.

- **DGain**

(Optional.) Controls the D gain of the PID controller.

- **IGain**

(Optional.) Controls the I gain of the PID controller.

- **PGain**

(Optional.) Controls the P gain of the PID controller.

- **val**

This slot is a generic pointer to a more specific slot describing the motor position, like **position** or **angle**, depending on the type of motor. It is mandatory in the Urbi Ready standard as a universal proxy to control an actuator. The more specific slot is described in a subclass of **Motor**.

26.5.8.1 LinearMotor

Subclass of [Interface.Motor](#). This interface is used to describe a linear motor controller.

A wheel can fall in this category, if the reported position is the distance traveled by a point at the surface of the wheel.

- **force**

Intensity of the measured or estimated force applied on a linear motor.

- **position**

Position of the motor in meters. Pointed to by the **val** slot.

26.5.8.2 LinearSpeedMotor

Subclass of [Interface.Motor](#). Motor similar to [Interface.LinearMotor](#), but controlled by its translation speed.

- **speed**

Translation speed in meters per second. Pointed to by the **val** slot.

26.5.8.3 RotationalMotor

Subclass of [Interface.Motor](#). This interface is used to describe a position-controlled rotational motor controller.

- **angle**
Angle of the motor in radian, modulo 2π . Pointed to by the `val` slot.
- **torque**
Intensity of the measured or estimated torque applied on the motor.
- **turn**
Absolute angular position of the motor, expressed in number of turns.

26.5.8.4 RotationalSpeedMotor

Subclass of [Interface.Motor](#). Interface describing a motor similar to `RotationalMotor` controlled by its rotation speed.

- **speed**
Rotation speed in radians per second.

26.5.9 Network

Contains information about the network identification of the robot.

- **IP**
IP address of the robot.

26.5.10 Sensor

This interface is used to describe a generic sensor.

- **val**
This slot is a generic pointer to a more specific slot describing the sensor value, like `distance` or `temperature`, depending on the type of sensor. It is mandatory in the Urbi Ready standard as a universal proxy to read a sensor. The more specific slot is described in a subclass of [Interface.Sensor](#).

26.5.10.1 AccelerationSensor

Subclass of [Interface.Sensor](#). This interface is used to describe an accelerometer.

- **acceleration**
Acceleration expressed in m/s^2 . Pointed to by the `val` slot.

26.5.10.2 DistanceSensor

Subclass of [Interface.Sensor](#). This interface is used to describe a distance sensor (infrared, laser, ultrasonic...).

- **distance**
Measured distance expressed in meters. Pointed to by the `val` slot.

26.5.10.3 GyroSensor

Subclass of [Interface.Sensor](#). This interface is used to describe an gyrometer.

- **speed**
Rotational speed in rad/s. Pointed to by the `val` slot.

26.5.10.4 Laser

Subclass of [Interface.Sensor](#). Interface for a scanning laser rangefinder, or other similar technologies.

- **angleMax**
End scan angle in radians, relative to the front of the device.
- **angleMin**
Start scan angle in radians, relative to the front of the device.
- **distanceMax**
Maximum measurable distance.
- **distanceMin**
Minimum measurable distance.
- **rate**
(Optional.) Number of scans per second.
- **resolution**
Angular resolution of the scan, in radians.
- **val**
Last scan result. Can be either a list of ufloat, or a binary containing a packed array of doubles.

Depending on the implementation, some of the parameters can be read-only, or can only accept a few possible values. In that case it is up to the implementer to select the closest possible value to what the user entered. It is the responsibility of the user to read the parameter after setting it to check what value will actually be used by the implementation.

26.5.10.5 TemperatureSensor

Subclass of [Interface.Sensor](#). This interface is used to describe a temperature sensor.

- **temperature**
Measured temperature in Celsius degrees. Pointed to by the `val` slot.

26.5.10.6 TouchSensor

Subclass of [Interface.Sensor](#). This interface is used to describe a touch pressure sensor (contact, induction, etc.).

- **pressure**
Intensity of the pressure put on the touch sensor. Can be 0/1 for simple buttons or expressed in Pascal units. Pointed to by the `val` slot.

26.5.11 SpeechRecognizer

Speech recognition allows to transform a stream of sound into a text using various speech recognition algorithms. Implementations should use the `micro` component as their default sound input.

- **`hear(s)`**

This event has one parameter which is the string describing what the speech engine has recognized (can be a word or a sentence).

- **`lang`**

(Optional.) The language used, in international notation (fr, en, it...): ISO 639.

26.5.12 TextToSpeech

Text to speech allows to read text using a speech synthesizer. Default implementations should use the `speaker` component (or alias) as their default sound output.

- **`age`**

(Optional.) Age of the speaker, if applicable.

- **`gender`**

(Optional.) Gender of the speaker (0:male/1:female).

- **`lang`**

(Optional.) The language used, in international notation ISO 639 (fr, en, it...).

- **`pitch`**

(Optional.) Voice pitch. A positive number, with 1 standing the regular pitch.

- **`say(s)`**

Speak the sentence given in parameter *s*.

- **`script(s)`**

(Optional.) Speak the text *s* augmented by script markups (see specific Gostai documentation) to generate Urbi events.

- **`speed`**

(Optional.) How fast the voice should go. A positive number, with 1 standing for “regular speed”.

- **`voice`**

(Optional.) Most TTS engines propose several voices, this attribute allows picking one. It's a string identifier specific to the TTS developer.

- **`voicexml(s)`**

(Optional.) Speak the text *s* expressed as a VoiceXML string.

26.5.13 Tracker

Camera-equipped robots can sometimes move the orientation of the field of view horizontally and vertically, which is a very important feature for many applications. In that case, this interface abstracts how such motion can be achieved, whether it is done with a pan/tilt camera or with whole body motion or a combination of both.

- **`pitch`**

Rotational articulation around the Y axis in the robot, expressed in radians.

- **`yaw`**

Rotational articulation around the Z axis in the robot, expressed in radians.

26.5.14 VideoIn

The VideoIn interface groups every information relative to cameras or any image sensor.

- **exposure**

(Optional.) Exposure duration, expressed in seconds. 0 if non applicable.

- **format**

(Optional.) Format of the image, expressed as an integer in the enum `urbi::UIImageFormat`. See below for more information.

- **gain**

(Optional.) Camera gain amplification (expressed as a coefficient between 0 and infinity). 1 if non applicable.

- **height**

Height of the image in the current resolution, expressed in pixels.

- **quality**

(Optional.) If the image is in the jpeg format, this slot sets the compression quality, from 0 (best compression, worst quality) to 100 (best quality, bigger image).

- **resolution**

(Optional.) Image resolution, expressed as an integer. 0 corresponds to the maximal resolution of the camera. Successive values correspond to all the supported image sizes in decreasing order. Once modified, the effective resolution in X/Y can be checked with the width and height slots.

- **val**

Image represented as a [Binary](#) value.

- **wb**

(Optional.) White balance (expressed with an integer value depending on the camera documentation). 0 if non applicable.

- **width**

Width of the image in the current resolution, expressed in pixels

- **xfov**

The x field of view of the camera expressed in radians.

- **yfov**

The y field of view of the camera expressed in radians.

The image sensor is expected to use the cheapest way in term of CPU and/or energy consumption to produce images of the requested format. Implementations linked to a physical image sensor do not have to implement all the possible formats. In this case, the format closest to what was requested must be used. A generic image conversion object will be provided. In order to avoid duplicate image conversions when multiple unrelated behaviors need the same format, it is recommended that this object be instantiated in a slot of the VideoIn object named after the format it converts to:

```
if (!robot.body.head.camera.hasSlot("jpeg"))
{
    var robot.body.head.camera.jpeg =
        ImageConversion.new(robot.body.head.camera.getSlot("val"));
    robot.body.head.camera.jpeg.format = 3;
}
```

urbiscript
Session

26.6 Standard Components

Standard components correspond to components typically found in wheeled robots, humanoid or animaloid robots, or in industrial arms. This section provides a list of such components. Whenever possible, robots compatible with the Gostai Standard Robotics API should name all the components in the hierarchy using this list.

26.6.1 Yaw/Pitch/Roll orientation

Note that Gostai Standard Robotics API considers the orientation to be a component name, and not a localizer. So one would write `head.yaw` and not `head.joint[yaw]` to refer to a rotational articulation of the head around the Z axis.

It is not always clear which rotational direction corresponds to the yaw, pitch or roll components (listed in the table below). This is a quick guideline to help determine the proper association.

Let us consider the robot in its resting, most prototypical position, like “standing” on two or four legs for a humanoid or animaloid, and let all members “naturally” fall under gravity. When gravity has no effect on a certain joint (because it is in the orthogonal plan to Z, for example), the medium position between rangemin and rangemax should be used. The body position achieved will be considered as a reference. Then for each component that is described in terms of yaw/pitch/roll sub-decomposition, the association will be as follow:

yaw rotational articulation around the Z axis in the robot.

pitch rotational articulation around the Y axis in the robot.

roll rotational articulation around the X axis in the robot.

When there is no exact match with the X/Y/Z axis, the closest match, or the default remaining available axis, should be selected to determine the yaw/pitch/roll meaning.

26.6.2 Standard Component List

The following table summarizes the currently referenced standard components, with a description of potential components that they could be sub-component of, a description of potential components they may contain, and a list of relevant interfaces. This table should be seen as a quick reference guide to identify available components in a given robot.

Name	Description	Sub. of	Contains	Facets
<code>robot</code>	The main component that represents an abstraction of the robot, and the root node of the whole component hierarchy.		<code>body</code>	Identity Network Mobile Tracker
<code>body</code>	The main component that contains every piece of hardware in the robot. This includes all primary kinematics sub-chains (arms, legs, neck, head, etc) and non-localized sensor arrays, typically body skin or heat detectors. Localized sensors, like fingertips touch sensors, will typically be found attached to the finger component they belong and not directly to the body.	<code>robot</code>	<code>wheel</code> <code>arm</code> <code>leg</code> <code>neck</code> <code>head</code> <code>wheel</code> <code>tail</code> <code>skin</code> <code>torso</code> ...	

continued on next page

Name	Description	Sub. of	Contains	Facets
wheel	Wheel attached to its parent component.	body		Rotational-Motor
leg	Legs are found in humanoid or animaloid robots and correspond to part of the kinematics chain that are attached to the main body by one extremity only and which do touch the ground in normal operation mode (unlike arms). A typical configuration for humanoids contains a hip, a knee and an ankle. If the leg is more segmented, the leg can be described with a simple array of joints.	body	hip knee ankle foot joint	
arm	Unlike legs, an arm's extremity does not always touch the ground in normal operating mode. This applies to humanoid robots or single-arm industrial robots. Arms supersede legs in the nomenclature: if a body part behaves alternatively like an arm and like a leg, it will be considered as an arm.	body	shoulder elbow wrist hand grip joint	
shoulder	The shoulder is the upper part of the arm. It can have one, two or three degrees of freedom and is the closest part of the arm relative to the body.	arm	yaw pitch roll	
elbow	Separates the upper arm and the lower arm, this is usually a single rotational axis.	arm	pitch	
wrist	Connects the hand and the lower part of the arm. Usually three degrees of freedom axis.	arm	yaw pitch roll	
hand	The hand is an extension of the arm that usually holds fingers. It's not the wrist, which is articulated and between the arm and the hand.	arm	finger	
finger	A series of articulated motors at the extremity of the arm, and connected to the hand. Fingers are usually localized with arrays and/or lateral localization respective to the hand.	hand	touch	Motor
grip	Simple two-fingers system.	arm hand	touch	Motor
hip	The hip is the upper part of the leg and connects it to the main body. It can have one, two or three degrees of freedom.	leg	yaw pitch roll	
knee	Separates the upper leg and the lower leg, this is usually a single rotational axis.	leg	pitch	
ankle	Connects the foot and the lower part of the leg. Usually three degrees of freedom axis.	leg	yaw pitch roll	
foot	The foot is an extension of the leg that usually holds toes. It's not the ankle, which is articulated and between the leg and the foot. The foot can also contain touch sensors in simple configurations.	leg	touch	

continued on next page

Name	Description	Sub. of	Contains	Facets
toe	Like fingers, but attached to a foot.	foot	touch	Motor
neck	The neck corresponds to a degree of freedom not part of the head, but relative to the rigid connection between the head and the main body.	body	yaw pitch roll	
tail	A series of articulated motors at the back of the robot.	body	joint	
head	The head main pivotal axis.	body neck	camera mouth ear lip eye eyebrow	
mouth	The robot mouth (open/close).	head	lip	Motor
ear	Ears may have degrees of freedom in certain robots.	head		Motor
joint	Generic articulation in the robot.	tail arm leg lip		Motor
yaw	Rotational articulation around the Z axis in the robot. See Section 26.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor
pitch	Rotational articulation around the Y axis in the robot. See Section 26.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor
roll	Rotational articulation around the X axis in the robot. See Section 26.6.1 .	body neck knee ankle shoulder elbow wrist torso		Rotational- Motor Rotational- Speed- Motor
x	Translational movement along the X axis.	body arm		Linear- Motor Linear- Speed- Motor
y	Translational movement along the Y axis.	body arm		Linear- Motor Linear- Speed- Motor

continued on next page

Name	Description	Sub. of	Contains	Facets
<code>z</code>	Translational movement along the Z axis.	<code>body arm</code>		<code>Linear-Motor</code> <code>Linear-Speed-Motor</code>
<code>lip</code>	Corresponds to animated lips.	<code>mouth</code>	<code>joint</code>	<code>Motor</code>
<code>eye</code>	Corresponds to the eyeball pivotal axis.	<code>head</code>	<code>camera</code>	
<code>eyebrow</code>	Some robots will have eyebrows with generally one or several degrees of freedom.	<code>head</code>	<code>joint</code>	<code>Motor</code>
<code>torso</code>	This corresponds to a pivotal or rotational axis in the middle of the main body.	<code>body</code>	<code>yaw pitch roll</code>	
<code>spine</code>	This is a more elaborated version of “ <code>torso</code> ”, with a series of articulations to give several degrees of freedom in the back of the robot.	<code>torso</code>	<code>joint</code>	
<code>clavicle</code>	This is not to be mixed up with the “top of the arm” body part. It is an independent degree of freedom that can be used to bring the two arms closer in a sort of “shoulder raising” movement.	<code>body</code>		<code>Motor</code>
<code>touch</code>	Touch sensor.	<code>finger</code> <code>grip</code> <code>foot</code> <code>toe</code>		<code>TouchSensor</code>
<code>gyro</code>	Gyrometer sensor.	<code>body</code>		<code>GyroSensor</code>
<code>accel</code>	Accelerometer sensor.	<code>body</code>		<code>Acceleration-Sensor</code>
<code>camera</code>	Camera sensor. If several cameras are available, localization shall apply; however there must always be an alias from <code>camera</code> to one of the effective cameras (like <code>cameraR</code> or <code>cameraL</code>).	<code>head body</code>		<code>VideoIn</code>
<code>speaker</code>	Speaker device. If several speakers are available, localization shall apply; however there must always be an alias from <code>speaker</code> to one of the effective speakers (like <code>speakerR</code> or <code>speakerL</code>).	<code>head body</code>		<code>AudioOut</code>
<code>micro</code>	Microphone devices. If several microphones are available, localization shall apply; however there must always be an alias from <code>micro</code> to one of the effective microphones (like <code>microR</code> or <code>microL</code>).	<code>head body</code>		<code>AudioIn</code>
<code>speech</code>	Speech recognition component.	<code>robot</code>		<code>Speech-Recognizer</code>
<code>voice</code>	Voice synthesis component.	<code>robot</code>		<code>TextTo-Speech</code>

continued on next page

Name	Description	Sub. of	Contains	Facets
------	-------------	---------	----------	--------

26.7 Compact notation

Components are usually identified with their full-length name, which is the path to access them inside the structure tree. For convenience and backward compatibility with pre-2.0 versions of Urbi, there is also a compact notation available. We will describe here how to construct the compact notation starting from the full name and the structure tree.

Full name	Compact name
<code>robot.body.armR.elbow</code>	<code>elbowR</code>
<code>robot.body.head.yaw</code>	<code>headYaw</code>
<code>robot.body.legL.knee.pitch</code>	<code>kneeL</code>
<code>robot.body.armR.hand.finger[3][2]</code>	<code>fingerR[3][2]</code>
<code>robot.body.armL.hand.fingerR</code>	<code>fingerLR</code>

The rule is to move every localization qualifier at the end of the compact notation, in the order where they appear in the full-length name. The remaining component names should then be considered one by one to see if they are needed to remove ambiguities. If they are not, like typically the robot or body components which are shared with almost every other full-length name, they can be ignored. If finally several component names have to be kept, they should be separated by using upper case letters for the first character instead of a dot, like in Java-style notation.

Some detailed examples:

- `robot.body.armL.hand.fingerR` gives `fingerLR`.
 1. Move all localization at the end: `robot.body.arm.hand.fingerLR`
 2. The full name remaining is: `robot.body.arm.hand.finger`
 3. `finger` should be kept, `hand`, `arm`, `body` and `robot` are not necessary since every finger component will always be attached only to a hand, itself attached to an arm and a body and a robot.
 4. The result is `fingerLR`
- `robot.body.head.yaw` gives `headYaw`.
 1. No localization to move
 2. `yaw` must be kept because `head` also have a `pitch` sub-component and
 3. `head` must also be kept to avoid ambiguity with other components having a `yaw` sub-component.
 4. The result is `headYaw`
- `robot.body.legL.knee.pitch` gives `kneeL`.
 1. Move all localization at the end: `robot.body.leg.knee.pitchL`
 2. `pitch` is not necessary because `knee` has only a `pitch`, so `knee` will be kept only
 3. The result is `kneeL`

26.8 Support classes

The Urbi SDK provides a few support urbiscript classes to help you build the component hierarchy. You can access to those classes by including the files ‘urbi/naming-standard.u’ and ‘urbi/component.h’.

26.8.1 Interface

The **Interface** class contains urbscript objects for all the interfaces defined in this document. Implementations must inherit from the correct interface.

```
// Instantiate a camera.
var cam = myCamera.new();
// Make it inherit from VideoIn.
cam.addProto(Interface.VideoIn);
```

urbscript
Session

The **Interface.interfaces** method can be called to get a list of all the interfaces an object implements.

26.8.2 Component

This class can be used to create intermediate nodes of the hierarchy. It provides the following methods:

- **addComponent(*name*)**
Add a new sub-component to the current component. *name* can be the name of the new component to create, or an instance of **Component**.
- **addDevice(*name*, *value*)**
Add device *value* as sub-component, under the name *name*. The device must inherit from at least one Interface.
- **dump**
Display a hierarchical view of the component hierarchy.
- **flatDump**
Display all the devices in the hierarchy, sorted by the Interface they implement.
- **makeCompactNames**
This function must be called once on the root node (**robot**) after the hierarchy is completed. It automatically computes the short name of all the devices, and insert them as slots of the **Global** object.

26.8.3 Localizer

The **Localizer** class is a special type of **Component** that stores other components based on their localization. It provides a '[]' operator that takes a **Localization**, such as **top**, **left**, **front**, and that can be used to set and get the **Component** or device associated with that **Localization**.

Note that the '[]' function is using a mechanism to automatically look for its argument as a slot of **Localizer**. As a consequence, you cannot pass a variable to this function, but only one of the constant **Localization**. To pass a variable, use the **get(loc)** or the **set(loc, value)** function.

The following example illustrates a typical instantiation sequence.

```
// Create the top-level node.
var Global.robot = Component.new("robot");
robot.addComponent("head");
var cam = MyCamera.new;
cam.addProto(Interface.VideoIn);
robot.head.addDevice("camera", cam);

// Add two wheels.
robot.addComponent(Localizer.new("wheel"));
robot.wheel[left] = MyWheel.new(0).addProto(Interface.RotationalMotor);
robot.wheel[right] = MyWheel.new(1).addProto(Interface.RotationalMotor);
```

urbscript
Session

```
// Implement the Mobile facet in urbiscript.
function robot.go (d)
{
    robot.wheel.val = robot.wheel.val + d / wheelRadius adaptive:1
};

function robot.turn(r)
{
    var v = r * wheelDistance / wheelRadius;
    robot.wheel[left].val = robot.wheel[left].val + v adaptive:1 &
    robot.wheel[right].val = robot.wheel[right].val - v adaptive:1
};
robot.addProto(Interface.Mobile);
robot.makeCompactNames;

// Let us see the result.
robot.flatDump;
[00010130] *** Mobile: robot
[00010130] *** RotationalMotor: wheelL wheelR
[00010130] *** VideoIn: camera
```

Part V

Urbi Platforms

About This Part

This part covers the specific features of Urbi for some of the platforms it was ported to. Environments not described in this part are covered in separate, stand-alone, documentations.

Chapter 27 — Aldebaran Nao

Nao is a humanoid robot Nao from Aldebaran Robotics.

Chapter 28 — Bioloid

Using the Bioloid robot construction kit with Urbi.

Chapter 29 — Mindstorms NXT

LEGO Mindstorms NXT is a programmable robotics kit released by Lego in July 2006, replacing the first-generation LEGO Mindstorms kit. The kit consists of 519 Technic pieces, 3 servo motors, 4 sensors (ultrasonic, sound, touch, and light), 7 connection cables, a USB interface cable, and the NXT Intelligent Brick.

Chapter 30 — Gostai Open Jazz

Gostai Open Jazz is an entirely programmable robot. It is shipped with the Urbi SDK to develop with Jazz.

Chapter 31 — Pioneer 3-DX

Pioneer 3-DX8 is an agile, versatile intelligent mobile robotic platform updated to carry loads more robustly and to traverse sills more surely with high-performance current management to provide power when it's needed.

Chapter 32 — Segway RMP

The Segway Robotic Mobility Platform is a robotic platform based on the Segway Personal Transporter.

Chapter 33 — Spykee

The Spykee is a WiFi-enabled robot built by Meccano (known as Erector in the United States). It is equipped with a camera, speaker, microphone, and moves using two tracks.

Chapter 34 — Webots

Using Cyberbotics' Webots simulation environment with Urbi.

Chapter 27

Aldebaran Nao

27.1 Introduction

The *Nao* is a 60 centimeters tall humanoid robot built by *Aldebaran* Robotics. It has an onboard Geode processor running Linux, 25 degrees of freedom, two onboard cameras, speakers, microphones, accelerometers, ultrasound and IR sensors...

27.2 Starting up

Nao comes with an installed version of Urbi.

On some versions of *Nao*, Urbi is not automatically started upon start-up. If this is the case, you must take the following steps to activate Urbi:

- Log in to your *Nao* using ssh: `ssh nao@myNao`
- Add a line containing `urbistarter` to '`/opt/naoqi/preferences/autoload.ini`'.
- Restart your *Nao*.

Those lines might already be present but commented out.

You should be able to connect to your *Nao* on the port 54000 and send it urbiscript:

```
tts.say("Hello, I am Nao."),
// Activate right arm.
armR.load = 1;
// Wave the arm: put it in position
shoulderRollR.val = -0.3 time: 0.5s |
shoulderPitchR.val = -1 speed: 0.9 |
// Wave it
timeout(6s) shoulderRollR.val = -0.4 sin:2s ampli:0.4;
// And put it back down, using the uncompressed name this time
robot.body.arm[right].shoulder.pitch.val = 2 speed:0.9;
// Activate all motors.
motors.on;
// Stand
motion.walkTo(0.1,0,0);
// Start walking...
tag: robot.walk(1),
// ... and interrupt the movement
sleep(2s) | tag.stop;
// Get the list of all devices and the interfaces they implement:
robot.dump;
```

urbiscript
Session

27.3 Accessing joints

All the Nao joints are accessible through their standard Nao and Urbi names, and respect the Urbi standard specifications for servo motors: Basically each motor has a *load* field that is linked to the motor stiffness, and a *val* field that can be used to read or write the motor position.

Some motor groups are also provided: they can be used exactly as joints, but any action performed on a group is sent to all the joints that are a member of this group.

urbiscript
Session

```
// Activate all motors
motors.load = 1;
// Or only all the head motors
head.load = 1;
// Move headYaw to 0.8 in one seconds. Since this commands takes
// one second to execute, terminate with a comma to be able
// to send other commands while it executes.
headYaw.val = 0.8 time: 1s,
// Move headPitch to -0.5 as fast as possible.
headPitch.val = -0.5;
// Move headYaw continuously in a sinusoidal trajectory...
tag: headYaw.val = 0 sin:2s ampli:0.5,
// ...and stop the movement
tag.stop;
```

The two hands are purposefully not included in the 'motors' group. Although you can control them using the *val* slot, it is recommended that you use the *open()* and *close()* methods, since they stop the motors once the movement is finished.

27.3.1 Advanced parameters

27.3.1.1 Trajectory generator period

The trajectory generators will issue write command at a period given by the `System.period` variable. The default is 10ms which is the native period of NaoQi.

27.3.1.2 Motor back-end method

Urbi can use two different methods to send the joint commands: DCM::`send()`, or ALMotor::`setAngle()`. The mode can be changed by calling `motor.setDCMWrite(bool)`.

In `setAngle()` mode, each write operation to a joint will synchronously call the `setAngle()` NaoQi method.

In DCM mode, Urbi will set a hook on the DCM update, and send all commands in this hook using the `send()` method. Urbi will send commands S milliseconds in the future. S defaults to 50ms, and can be changed by calling `ALMotor.setDCMShift(shiftValue)`.

27.3.1.3 Motor command debugging

The motors implements a debugging mode that can be activated by calling `ALMotor.setTrace(1)`. While activated, it will write all sent commands to the file '/tmp/traj.log'. Each line will contain the motor number, the command timestamp, and the command position. Be careful not to let this feature active while not using it, as it would quickly fill-up the memory.

27.4 Leds

All Nao's leds and led groups, as defined by NaoQi, are available as objects in urbiscript. You can set the led values by writing to the slots *val* (intensity between 0 and 1), *r*, *g*, *b* (color intensity between 0 and 1) or *rgb* (RGB value, one byte each, i.e. 0xFF0000 is red).

27.5 Camera

The Nao camera object is instantiated by default, under the standard name `camera`. It is deactivated by default, set its `load` field to 1 to activate. The default frame-rate is only 4fps, so you might want to increase it before activating the camera.

If you wish to use RTP instead of TCP to receive the camera image, you must:

- load the RTP module in the engine by typing:

```
loadModule("urbi/rtp");
```

urbiscript
Session

- also load the RTP module with your remote UObject by adding it to your urbi-launch command line.

RTP will then be used automatically.

27.5.1 Slots

- **format**

Set the image format: 1 for a jpeg-compressed image, 0 for raw yuv.

- **formatDetail**

Set the image format, providing more options:

- 1 raw RGB.
- 2 raw YUV.
- 3 JPEG.
- 4 RGB with a PPM header.
- 5 YUV422: YUYV, 2 bytes/pixel.
- 6 Grey8: gray image, 1 byte/pixel.
- 7 Grey4: gray image 4 bits/pixel.

JPEG mode gives the smallest images and uses less bandwidth to access the images remotely, but uses more CPU power.

- **quality**

Set JPEG compression quality, from 0 to 100.

- **rate**

Set image frame rate in Hz.

- **resolution**

Set image resolution. 0 gives the max resolution (640x380), 1 divides each dimension by two(320x240), 2 divides by four(160x120), 3 by eight(80x60).

- **width**

Image width. Read-only.

- **height**

Image height. Read-only.

- **threaded**

Set to 1 to activate threaded mode, and to 0 to deactivate. This must be set before switching load to 1 for the first time. Defaults to 1.

27.6 Whole body motion

The whole body motion component of the NaoQi can be activated by reading and writing to the `pos` slot of both hands. This sets a target position in Cartesian space for the hand. The NaoQi will use its whole body to try to reach it. Be careful to only ask for reachable positions.

27.7 Other sensors

The following sensors are available through a urbiscript variable. Reading the variable will give the latest available sensor value.

- `sonarL.val, sonarR.val`
last left and right value from the Sonar module.
- `footTouchL.val, footTouchR.val`
Foot bumper, can be 0 or 1.
- `headTouchR.val, headTouchM.val, headTouchF.val`
Head touch sensor segments.
- `accelX.val, accelY.val, accelZ.val`
Accelerometer values.
- `gyroX.val, gyroY.val`
Gyroscope values.
- `battery.voltage, battery.current`
Current battery voltage and current usage.
- `fsr.val, fsr.left, fsr.right`
1 if both, the left and the right foot are touching the ground, respectively.
- `fsr.leftFootTotalWeight, fsr.rightFootTotalWeight`
Weight under each feet.

27.8 Interfacing with NaoQi

27.8.1 Accessing the NaoQi shared memory region

Many NaoQi modules communicate through a shared hash table handled by the `ALMemory` module. This module is available under the `stm` name in Urbi, and has the following slots:

- `get(name)`
Return the value of memory location `name`.
- `set(name, value)`
Set memory location `name` to `value`.
- `bindRenameVariable(memoryName, variableName)`
Create a urbiscript slot named `variableName`, and synchronize it with memory location `memoryName`. Only read support is available. `variableName` can be of the form "a.b", in which case it will create variable "b" in object "a", or of the form "b", in which case variable "b" will be created in object "stm".

27.8.2 Accessing standard NaoQi modules

All standard NaoQi modules (ALMemory, ALLogger, ALMotion, ALFrameManager, ALTextToSpeech, ALAudioPlayer...) are available in Urbi. You can call all their methods directly.

27.8.3 Binding new NaoQi modules in Urbi

You can create a proxy on every NaoQi module, local or remote. The function `ALProxy` can be used to ease the process:

Here is an example with red ball detection and tracking module:

```
// Instantiate a proxy to RedBallDetection
var Global.redBallDetection = ALProxy("RedBallDetection", [], []);
// Instantiate a proxy to RedBallTracker
var Global.redBallTracker = ALProxy("RedBallTracker", [], []);
// Enable nao head motors
robot.body.head.load = 1;
// Launch the red ball tracking
redBallTracker.startTracker ();
sleep(60s);
// Stop red ball tracking
redBallTracker.stopTracker ();
// reset head position
headYaw.val = 0 speed: 0.8 & headPitch.val = 0 speed: 0.8;
```

urbiscript
Session

Another example with the vision toolbox module:

```
var vision = ALProxy("VisionToolbox", [], ["isItDark",
    "takePicture", "takePictureRegularly", "setWhiteBalance",
    "startVideoRecord", "startVideoRecord_adv"]);
switch(vision.isItDark)
{
    case var x if x <= 4:
        echo("We are outdoor");
    case var x if x <= 100:
        echo("We are indoor, it's bright");
    case var x if x > 100:
        echo("Here is a shadowed place");
};
```

urbiscript
Session

The first argument to `ALProxy` is the name of the module without the "AL". The second argument is a list of alternate names for the module. The third argument is a list of method names that must be called asynchronously. You must pass in this list all the functions that takes a long(more than a few milliseconds) time to execute. Synchronous calls have less overhead, but will freeze the urbiscript interpreter for the duration of the call.

Most of the standard modules are loaded by the initialization script 'URBI.INI'.

27.8.4 Writing NaoQi modules in Urbi

You can create independent modules capable of answering requests and services from other NaoQi modules, written in C++, urbiscript or ruby. Let's see an example:

```
// -----
// Simple module creation samples
// -----
// the name of your variable and of the module must be the SAME!
var urbiHelloWorld = ALModule.new("urbiHelloWorld");

// create a new method doing some interesting things
function urbiHelloWorld.sayHello()
{
    // Use ALTextToSpeech, instantiated by URBI.INI
    tts.say("Hello");
};

// inform the world that there's a new method, and what it's doing
urbiHelloWorld.bindUrbiMethod("sayHello", "Nao will say 'Hello'", [], []);

// to test it outside of Urbi, you can for example type in a browser:
```

urbiscript
Session

```
// http://<Nao's IP>:9559/?eval=urbiHelloWorld.version(); or
// http://<Nao's IP>:9559/?eval=urbiHelloWorld.ping(); and more naturally
// http://<Nao's IP>:9559/?eval=urbiHelloWorld.sayHello();

// -----
// another module example
// -----
var mathematics = ALModule.new("mathematics");

function mathematics.add(a, b)
{
    var res = a + b;
    echo("mathematics.add => " + res);
    return res;
};

mathematics.setModuleDescription(
    "A powerful module to make remote computing in urbiscript :)");
mathematics.bindUrbiMethod("add", "Compute the sum of two numbers",
    [{"a": "first number of the addition"}, {"b": "second number of the addition"}], ["sum", "the sum of the addition"]);

// to test it outside of urbi, you can for example enter in a browser:
// http://<Nao's IP>:9559/?eval=mathematics.add(2,3)
```

27.8.5 More examples

Here is an example that shows how to declare in urbiscript a callback called by a NaoQi module:

urbiscript
Session

```
// Instantiate a proxy to RedBallDetection
var redBallDetection = ALProxy("RedBallDetection", [], []);
// Instantiate a proxy to RedBallTracker
var redBallTracker = ALProxy("RedBallTracker", [], []);
// Create an urbi ALModule that contains a callback function
var urbiRedBall = do (ALModule.new("urbiRedBall"))
{
    function ballDetected(varName, value, message) {
        echo("BallDetected was triggered. Got: %s, %s, %s" % [varName, value, message]);
    };
    bindUrbiMethod("ballDetected", "called when ball is detected", [], []);
};
// Trigger the callback function whenever a ball is detected.
memory.subscribeToMicroEvent("redBallDetected", "urbiRedBall",
    "ballisdetected", "ballDetected");
// Launch the red ball tracking
redBallTracker.startTracker();
```

Playing in urbiscript with nao eyes:

urbiscript
Session

```
class Global.EyeLeds
{
    function rotate(var inc, var r, var g, var b,
                  var frequency = 80ms, var duration = 3)
    {
        var eye_ = this.led.asList;
        var i = 0; var p = -inc; var j = inc;
        timeout (duration)
            every (frequency)
            {
                i = (i+inc)%8; p = (p+inc)%8; j = (j+inc)%8;
                eye_[p].val = 0;
                eye_[i].r = r; eye_[i].g = g; eye_[i].b = b;
                eye_[j].r = r; eye_[j].g = g; eye_[j].b = b;
```

```

    };

};

function redRotate() { rotate(1, 1, 0, 0); };
function greenRotate() { rotate(1, 0, 1, 0); };
function blueRotate() { rotate(1, 0, 0, 1); };

function randomize(var duration = 3)
{
    timeout(duration)
    loop
    {
        led.r = random(100)/100;
        led.g = random(100)/100;
        led.b = random(100)/100;
    };
};

function redBlink(var duration = 3) {
    timeout(duration) led.r = 0.5 sin:1s ampli:0.5;
};
function greenBlink(var duration = 3) {
    timeout(duration) led.g = 0.5 sin:1s ampli:0.5;
};
function blueBlink(var duration = 3) {
    timeout(duration) led.b = 0.5 sin:1s ampli:0.5;
};

function color(var r, var g, var b) {
    led.r = r; led.g = g; led.b = b;
};

function red() { color(1, 0, 0); };
function green() { color(0, 1, 0); };
function blue() { color(0, 0, 1); };
function black() { color(0, 0, 0); };

function __unit(slot)
{
    eyeR.black & eyeL.black;
    voice.say(slot) &
    eyeR.getSlot(slot).apply([eyeR]) &
    eyeL.getSlot(slot).apply([eyeL]);
};

function __test()
{
    voice.say("Eyes test procedure");
    __unit("red");
    __unit("green");
    __unit("blue");
    __unit("redBlink");
    __unit("greenBlink");
    __unit("blueBlink");
    eye.black;
    voice.say("blink yellow") &
    eyeL.greenBlink & eyeL.redBlink &
    eyeR.greenBlink & eyeR.redBlink;
    __unit("randomize");
    __unit("redRotate");
    __unit("greenRotate");
    __unit("blueRotate");
    eye.black;
};
};

```

```

do (eyeL)
{
    addProto(EyeLeds);

    function rotateRight(var r, var g, var b, var frequency = 50ms,
                        var duration = 1s) {
        rotate(-1, r, g, b, frequency, duration)
    };

    function rotateLeft(var r, var g, var b, var frequency = 50ms,
                        var duration = 1s) {
        rotate(1, r, g, b, frequency, duration)
    };
};

do (eyeR)
{
    addProto(EyeLeds);

    function rotateRight(var r, var g, var b, var frequency = 50ms,
                        var duration = 1s) {
        rotate(1, r, g, b, frequency, duration)
    };

    function rotateLeft(var r, var g, var b, var frequency = 50ms,
                        var duration = 1s) {
        rotate(-1, r, g, b, frequency, duration)
    };
};

EyeLeds.__test;

```

This example shows how to set nao into its initial standing pose:

urbiscript
Session

```

var HeadYawAngle      = 0;
var HeadPitchAngle    = 0;
var ShoulderPitchAngle = 80;
var ShoulderRollAngle = 20;
var ElbowYawAngle     = -80;
var ElbowRollAngle    = -60;
var WristYawAngle     = 0;
var HandAngle          = 0;
var HipYawPitchAngle  = 0;
var HipRollAngle       = 0;
var HipPitchAngle      = -25;
var KneePitchAngle     = 40;
var AnklePitchAngle   = -20;
var AnkleRollAngle     = 0;
var Head = [];
var LeftArm = [];
var RightArm = [];
var LeftLeg = [];
var RightLeg = [];

switch (ALMotion.getRobotConfig[1][0])
{
    case "naoAcademics":
        Head      = [HeadYawAngle, HeadPitchAngle];
        LeftArm  = [ShoulderPitchAngle, +ShoulderRollAngle,
                   +ElbowYawAngle, +ElbowRollAngle, WristYawAngle, HandAngle];
        RightArm = [ShoulderPitchAngle, -ShoulderRollAngle,
                   -ElbowYawAngle, -ElbowRollAngle, WristYawAngle, HandAngle];
        LeftLeg  = [HipYawPitchAngle, +HipRollAngle,
                   HipPitchAngle, KneePitchAngle, AnklePitchAngle, +AnkleRollAngle];
        RightLeg = [HipYawPitchAngle, -HipRollAngle,
                   HipPitchAngle, KneePitchAngle, AnklePitchAngle, -AnkleRollAngle];
}

```

```

RightLeg = [HipYawPitchAngle, -HipRollAngle,
            HipPitchAngle, KneePitchAngle, AnklePitchAngle, -AnkleRollAngle]

case "naoRobocup":
    Head      = [HeadYawAngle, HeadPitchAngle];
    LeftArm   = [ShoulderPitchAngle, +ShoulderRollAngle,
                +ElbowYawAngle, +ElbowRollAngle];
    RightArm  = [ShoulderPitchAngle, -ShoulderRollAngle,
                -ElbowYawAngle, -ElbowRollAngle];
    LeftLeg   = [HipYawPitchAngle, +HipRollAngle,
                HipPitchAngle, KneePitchAngle, AnklePitchAngle, +AnkleRollAngle];
    RightLeg  = [HipYawPitchAngle, -HipRollAngle,
                HipPitchAngle, KneePitchAngle, AnklePitchAngle, -AnkleRollAngle];

case "naoT14":
    Head      = [HeadYawAngle, HeadPitchAngle];
    LeftArm   = [ShoulderPitchAngle, +ShoulderRollAngle,
                +ElbowYawAngle, +ElbowRollAngle, WristYawAngle, HandAngle];
    RightArm  = [ShoulderPitchAngle, -ShoulderRollAngle,
                -ElbowYawAngle, -ElbowRollAngle, WristYawAngle, HandAngle];

case "naoT2":
    Head      = [HeadYawAngle, HeadPitchAngle];
};

// Gather the joints together
var pTargetAngles = Head + LeftArm + LeftLeg + RightLeg + RightArm;
// Convert to radians
pTargetAngles = pTargetAngles.map(function(x) {x*motion.TO_RAD});

// ----- send the commands -----
// We use the "Body" name to denote the collection of all joints.
var pNames = "Body";
// We set the fraction of max speed
var pMaxSpeedFraction = 0.2;

ALProxy("RobotPose");
switch(ALRobotPose.getActualPoseAndTime)
{
    case "crouch":
        // Ask motion to do this with a blocking call
        ALMotion.angleInterpolationWithSpeed(pNames, pTargetAngles, pMaxSpeedFraction);
    case "stand":
        // Ask motion to do this with a blocking call
        ALMotion.angleInterpolationWithSpeed(pNames, pTargetAngles, pMaxSpeedFraction);
    default:
        voice.setLanguage("english");
        voice.say("Please first stand me up");
};

```


Chapter 28

Bioloid

28.1 Introduction

The *Bioloid* is a robot construction kit made of servomotors, sensors and frame elements. You can find more information on the manufacturer's official website at http://www.robotis.com/zbxe/bioloid_en.

Urbi cannot run directly on the Bioloid controller (CM5, CM510). Urbi runs on your computer and talks to the Bioloid controller over serial link (RS232, or wireless depending on your configuration). Urbi can also talk directly to the motor bus if you have an usb2dynamixel.

28.2 Installing Urbi for Bioloid

28.2.1 Flashing the firmware

Urbi for Bioloid is using a custom firmware in the CM-5 controller. You must upload the new firmware in your CM-5 using the procedure below. This operation is reversible.

Download the firmware file from our website at <http://www.gostai.com/downloads/urbi-for-bioloid.zip>.

- Start the *robot terminal* software from Robotis (installed with your Bioloid software).
- Make sure your CM-5 batteries are fully charged or that it is plugged to a power supply.
- Connect your CM-5 to the serial port of your computer.
- Turn off your CM-5 (using the red on/off button).
- Maintain the '#' key pressed on your keyboard while turning the CM-5 back on. A message like this one should be displayed:

```
SYSTEM O.K. (CM5 boot loader V1.xx)
-#####
```

- Press enter, type load and press enter again. Display should be:

```
SYSTEM O.K. (CM5 boot loader V1.xx)
-#####
-load
Write Address: 00000000
Ready..
```

- In the menu, select “Files”, “Transmit file” (Control-T) and select the firmware file you downloaded. After a few seconds, display should turn to:

```
Ready..Success
(and other information)
```

28.2.2 Getting Urbi and Urbi for Bioloid

Look at <http://www.urbiforge.org/index.php/Robots/Bioloid> for up-to-date information on how to download Urbi for Bioloid.

28.3 First steps

28.3.1 Starting up

Connect your CM-5 to the PC and turn it on before starting Urbi.

When you start Urbi, the initialization script in ‘global.u’ will connect to the CM-5 (edit the file to change the port if required), scan for devices and instantiate them. It will create:

- one `bioloid` object representing the connection to the CM5.
- one object per motor (`AX12`) named `motorid` and `motors[id]` where `id` is the motor identifier.
- one object per sensor (`AXS1`) named `sensorid` and `sensors[id]` where `id` is the sensor `id` (e.g., `sensor100`, `sensors[100]`).

Once initialized, do not disconnect any motor that was detected or everything will run very slow.

At this point you might want to give motor names more adapted to your model. Here is an example:

```
urbiscript
Session
do (Global) // Make them available to everyone.
{
    var wheelL = motors[1]; // Left wheel is motor ID 1.
    var wheelR = motors[2]; // Right wheel is motor ID 2.
    var headYaw = motors[7]; // Head yaw rotation is motor ID 7.
};
```

28.3.2 Motor features

All the AX12 features are exposed in urbiscript. The following section lists the main slots with code examples explaining how to use them. You can use `Object.localSlotNames` and refer to the AX12 documentation for more information.

- `val`

The current motor position when read, the target position when written to.

```
urbiscript
Session
// Move motor6 to 90 degrees.
motor6.val = 90deg;
// Move to 0 in 5 seconds.
motor6.val = 0 time:5s;
// Ticking clock.
at (motor6.val > 0) echo("tick") onleave echo("tack");
motor6.val = 0 sin:1s ampli:20deg,
```

- `cwLimit, ccwLimit`

Set min and max reachable angles. When both equal to 0, the motor is put in continuous rotation mode. In this mode, writing to `val` has no effect: the motor moves at the speed given by the `speed` slot.

- `speed`

When read, gives the current rotation speed. When written to, set the speed at which following commands will be executed. In continuous rotation mode, start moving the motor at given speed.

```
// Set speed to two radians/second.
motor6.speed = 2;
// Move to 90deg.
motor6.val = 90deg;
// Switch to continuous rotation mode.
motor6.cwLimit = motor6.ccwLimit = 0;
// Start moving counter-clockwise at 1 radian/seconds.
motor6.speed = 1;
```

- **torque**

Current torque the motor is giving.

- **load**

Shut down the motor when 0.

```
//Turn the motor off if the torque is too high.
at(motor6.torque > 5) motor6.load = 0;
```

urbiscript
Session

28.3.3 Sensor features

All the AXS1 features are exposed in urbiscript. The following section lists the main slots with code examples explaining how to use them. You can use `Object.localSlotNames` and refer to the AXS1 documentation for more information.

- **IRLeft, IRCenter, IRRight**

Amount of infrared received by the sensor using the internal emitter. This sensors detect light reflected by objects nearby in the direction of the sensor.

- **lightLeft, lightCenter, lightRight**

Amount of infrared received by the sensor without the emitter. Detects infrared light sources such as incandescent light, candles, ...

- **buzzerIndex**

Play a note when written to.

- **buzzerTime**

Set length of next played note.

- **clapCount**

Number of successive claps detected by the microphone.

- **soundVolume**

Volume of sound received by the microphone.

```
// The louder the sound, the faster we go.
at (sensor100.soundVolume->changed?)
    motor6.speed = sensor100.soundVolume * 6;

// Stop after 4 claps.
at (sensor100.clapCount == 4)
    motor6.speed = 0;
```

urbiscript
Session

- **soundVolumeMax**

Holds maximum sound volume recorded so far. Write 0 to this slot to reset.

Chapter 29

Mindstorms NXT

This documentation contains information about the LEGO Mindstorms NXT Urbi engine. The first two sections provide a short tutorial for using Urbi to control the TriBot robot. The remainder of the documentation contains an exhaustive reference for the LEGO Mindstorms NXT Urbi engine devices.

29.1 Launching Urbi for Mindstorms NXT

First you should build the TriBot model from the LEGO Mindstorms NXT basic model.

Install the engine Mindstorms by untaring it into the Urbi engine runtime folder. Follow instructions in ‘INSTALL’ and ‘README’ files.

The file ‘TriBot.u’ is loaded by default when server is launched. You can remove it by modifying ‘global.u’.

29.2 First steps with Urbi to control Mindstorms NXT

This section is a short tutorial for using Urbi with the LEGO TriBot model with the default configuration provided with the Urbi server. Users familiar with Urbi should look directly to the next section for an extensive list of the available devices.

29.2.1 Make basic movements

We will first move the wheels of the robot.

In Urbi, all the motor’s speed and the sensor’s value in a robot are associated with variables. You can set the motor’s speed or get the sensor’s value by assigning or reading the values of the corresponding variables.

In the default server layout for the TriBot model, the left wheel is assigned to variable `wheelL` and the right wheel is assigned to `wheelR1`. The values of these variables control the corresponding wheel speed. First put the TriBot upside-down to avoid any accident, then you can move the left wheel just doing:

```
wheelL.speed = 50;
```

urbiscript
Session

Don’t omit the semicolon at the end of the line, or the command will not be executed. You can also move the right wheel (a negative value is a move backward):

```
wheelR.speed = -50;
```

urbiscript
Session

A group is also defined so that you can give orders to several motors at the same time. In our case, the group `wheels` contains both robot wheels. Setting `wheels` to a value will set both wheels to this value:

¹To be precise, sensors and motors are associated with objects. The left motor is associated with the `wheelL` object and the left motor speed is associated with the `wheelL.speed` slot of this object.

```
wheels.speed = 0;
```

will stop both wheels. The third motor of the TriBot is associated with the claw variable.

29.2.2 Improving the movements

The commands given so far simply set a speed to the wheels, but you can use Urbi to make precise sequences of movements. It is for example possible to set a speed for a given time, then stop.

To achieve that, Urbi provides a command to wait for a given time: sleep (duration). And you can make sequences of commands separated with semicolon that will be executed one after another. So the following line:

urbiscript
Session

```
wheels.speed = 50;  
sleep(10s);  
wheels.speed = 0;
```

will make the robot to go forward during 10 seconds then stop.

It is also possible with Urbi to control the way the values will change. In the previous examples, the motor speed goes from 0 to 50 as fast as possible. You can use a modifier to the variable assignation to control the way the variable value will change. For example, if you want to reach a value in a given time, use the time: modifier:

urbiscript
Session

```
wheels.speed = 50 time:3s;  
wheels.speed = 0 time:3s;
```

Using this code, the robot will accelerate to reach the speed of 50 after 3 seconds, then slow down to stop after another 3 seconds. Other modifiers are available in Urbi, described in the Urbi documentation.

29.2.3 Reading sensors

In all the previous steps the commands don't take the environment into account. To do that, you need to get the sensor values. In Urbi, you just have to type the associated variable name. For example when you type:

urbiscript
Session

```
sonar.val;
```

The server returns a message with the ultrasonic sensor value:

urbiscript
Session

```
[00146711] 48
```

In this message, 146711 is the time (from the server clock) at which the value was read, 48.00000 is the value of the sonar sensor, in this case the distance to the obstacle in front of it. You also need a way to react to the sensor values. In Urbi, this is achieved using event catching commands. These commands check some condition and launch other commands when these conditions are verified. For example the following command:

urbiscript
Session

```
at (sonar.val < 50)  
wheels.speed = 0;
```

will stop the robot if an obstacle is detected at less than 50 centimeters. Note that this command remains active in the server. If you move the robot and set a forward speed:

urbiscript
Session

```
wheels.speed = 50;
```

the robot will stop again when another obstacle is encountered. The other sensors available for the TriBot are the sound sensor decibel, the bumper sensor bumper and the light sensor light. The following command:

urbiscript
Session

```
at (bumper.val == 1)
wheels.speed = -50;
```

will make the robot go backward if something hits the bumper. Note that the `at` command reacts only when the condition becomes true. If you want to loop the execution of an action whenever the condition is true, use the `whenever` command (see [Section 29.2.5](#)).

29.2.4 Tagging commands

The `at` commands you entered before are still active in the server. You need a way to stop them if you want to do something else. The tagging mechanism will enable you to do that. A tag in a name associated to a command. To tag a command with the name `myTag`, create it and simply prefix the command. For example:

```
var myTag = Tag.new|;
myTag: at (decibel.val > 0.7)
wheels.speed = 0;
```

urbiscript
Session

will launch the event catching command using the decibel sensor with tag `myTag`. Later you will be able to stop that command:

```
myTag.stop;
```

urbiscript
Session

will stop the previous command and the robot will not react to sound anymore. It is also possible to freeze temporarily a command:

```
myTag.freeze;
```

urbiscript
Session

will stop the commands tagged with `myTag`, but it will not be deleted, and

```
myTag.unfreeze;
```

urbiscript
Session

will resume the command.

29.2.5 Playing sounds

The LEGO Mindstorms NXT is able to play sounds. The object `beeper` is associated with this capacity. However, you can't just assign a value to the variable `beeper`, because playing a sound requires several parameters. You therefore have to call a method of the `beeper` object:

```
beeper.play(200, 3s);
```

urbiscript
Session

This will play a beep at 200Hz during 3 seconds (beep's frequency must be between 200Hz and 14000Hz). The parameters of the method may also be the result of operations, or depend on other variables. For example, it is possible to play a sound whose frequency depends on the obstacle distance:

```
var myTag = Tag.new|;
myTag: whenever (sonar.val < 100)
beeper.play(200 + 6 * sonar.val, 3ms);
```

urbiscript
Session

This plays beeps with a lower frequency as the obstacle comes closer (under 1 meter).

29.2.6 Cyclic moves

We now introduce a modifier in Urbi that makes it possible to do cyclic moves of sinusoidal shape. This modifier has the particularity to make the assignment never terminate. To be able to enter new commands after this modifier is used, you need to put this command in background so that the Urbi server continues processing new commands. This is achieved by terminating the command with a comma instead a semicolon. More details about these command separators will be given in [Section 29.2.7](#). The `sinus` modifier can be used this way:

urbiscript
Session

```
var myTag = Tag.new();
myTag: wheels.speed = 0 sin:10s ampli:100,
```

This command will make the speed of the wheels oscillate around 0, between -100 et 100 with a period of 10s. Use a tag to be able to stop the command. To make the speed change from 0 to 100 with a period of 3 seconds, use the command:

urbiscript
Session

```
var tag2 = Tag.new();
tag2: wheels.speed = 50 sin:3s ampli:50,
```

29.2.7 Parallelism

Urbi handles command parallelism in a very simple way. Parallelism is enforced through various command separators:

- As usual in programming languages, two commands separated by a semicolon will be executed one after the other:

urbiscript
Session

```
wheels.speed = 100;
sleep(2s);
wheels.speed = 0;
```

will turn the wheels, wait 2 seconds and then stop.

- In Urbi, two commands separated by & will be executed in parallel and will start exactly at the same time:

urbiscript
Session

```
wheelL.speed = 50 & wheelR.speed = -50;
```

will move at the same time the left wheel forward and the right wheel backward (so the robot will turn).

- The comma makes the first action to go in background and starts the next one as soon as possible, without enforcing a simultaneous start of the two commands:

urbiscript
Session

```
wheelL.speed = 50 time:10s,
wheelR.speed = -40 time:10s,
```

will not wait that wheelL grows to 50 to make wheelR decrease to -40, but will not enforce that the commands terminate at the same time, which would be the case with the & separator.

This explain why the comma was necessary in the previous section. If we finish a command that last forever with a semicolon, the server will wait forever before starting a new command. If you happen to make this mistake, it is still possible to open a new connection to the Urbi server using another client and using the stop tag; command.

Here is an example where we use parallelism to move the TriBot awkwardly while playing sounds:

urbiscript
Session

```
wheelR.speed = 0 sin:3s ampli:50 &
wheelL.speed = 0 sin:4s ampli:30 &
beeper.play(200,3s),
```

Parallelism handling is a key feature of Urbi, which has in fact four command separators.

29.2.8 Using functions

As in most programming language, it is possible to write functions in Urbi. The functions are useful for making more complex programs where the same commands are used several times. The following example defines a function which moves the robot for a given time at a given speed:

```
function Global.forward(speed,timer)
{
    wheels.speed = speed;
    sleep(timer);
    wheels.speed = 0;
},
```

urbiscript
Session

Once defined, this function can be called simply:

```
Global.forward(100,3000);
```

urbiscript
Session

More details about functions, objects and variables in Urbi are available in the Urbi tutorial.

29.2.9 Loading files

When making larger Urbi programs, it is easier to save them in files and to command the server to load the files. The files are simple text files in which you write Urbi commands. These files should be saved in the data/ sub-directory of your sever directory. The files should use the .u extension which is the extension used for Urbi script files. An example ‘**demo.u**’ is provided with the server. To make the Urbi server to execute this script, type the Urbi command:

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

urbiscript
Session

29.2.10 Conclusion

This chapter has highlighted the main features of Urbi with the Mindstorms TriBot robot. The next chapter details all the devices available for the TriBot in the default layout.

29.3 Default layout reference

This chapter describes the objects defined in the ‘TriBot.u’ layout shipped with the Urbi Mindstorms NXT server.

29.3.1 Motors

Three motors are defined: `wheelL`, `wheelR` and `claw`. A group, `wheels`, is defined to group `wheelL` and `wheelR`:

```
wheels.speed = 10;
```

urbiscript
Session

is equivalent to

```
wheelL.speed = 10 & wheelR.speed = 10;
```

urbiscript
Session

The slot `.speed` allows you to set the current speed of the motor. This value is set between -100 and 100.

Motor position (in degrees) can be accessed using the `.val` slot. This slot is set to 0 when you turn the robot on.

29.3.2 Sensors

Sensors are grouped in the sensors group and all hardware devices (sensors + motors + battery + beeper) are grouped in the hardware group.

29.3.2.1 Bumper

The switch device in front of the robot is called bumper. Its value is 1 if it is pressed, 0 otherwise.

29.3.2.2 Sonar

The ultrasonic sensor is called sonar. Its value is the distance measured by the sensor in centimeters between 0 and 255. When the read operation fails, 255 is returned.

29.3.2.3 Decibel

The sound sensor is called decibel. Its value relates the level of ambient sound. It is between 0 and 1. Two different modes can be used by changing the .mode slot to "DB" or "DBA".

DBA is a mode measuring only frequencies between 200 and 14000Hz.

DB measures a wider band.

Default value is "DBA".

29.3.2.4 Light

The light sensor is called light. The value returned is between 0 and 1 representing the amount of light measured. Three different modes are available by changing the .mode slot.

Reflector means the sensor lights on its led and measures the light reflected.

Ambient lights off the led and measures the ambient light.

Normal lights off the led too and return the raw value measured.

Default value is "Reflector".

29.3.3 Battery

The battery device is called battery. Its value is the current battery level between 0 and 1, 1 is full charge.

29.3.4 Beeper

The beeper device is called beeper. You can request the beeper to play a sound using the play method:

urbiscript
Session

```
beeper.play(frequency, time);
```

frequency is an integer between 200 and 14000, which is the frequency of the sound. **time** is an integer, which is the duration of the sound in milliseconds. A duration of 0 is infinite. The command returns immediately. If you want to wait until the end of the beep, use:

urbiscript
Session

```
{
  beeper.play(frequency, duration);
  sleep(duration)
},
```

29.3.5 Command

The Command UObject is not a device. It makes it possible to send direct commands as you do with the NXT SDK. You will find more information on <http://mindstorms.lego.com/>. This object has methods to send and receive data to the NXT:

```
Command.send(buff);
```

urbiscript
Session

where:

- *buff* is a valid command buffer (list of integers between 0 and 255).

For example:

```
Command.send([3, 10, 10, 0, 0]);
```

urbiscript
Session

will play a beep.

```
answer = Command.request(buff, size) ;
```

urbiscript
Session

where:

- *buff* is a valid command buffer (requiring an answer)
- *size* is the size of the return buffer (it must be the exact size)
- *answer* is a list containing the values of the return buffer ([] is returned when the request failed)

This command is designed for expert users. Using it is not recommended if you don't know what you are doing and might result in server crashes. Here is an example:

```
answer = Command.request([7,0], 15);
```

urbiscript
Session

that returns the information on input port 1.

29.4 How to make its own layout

In this chapter, the method to build a custom layout for a different robot is explained. A good understanding of the objects in Urbi in useful and the reading of the Urbi tutorial is advised.

There are three kinds of devices provided by the Mindstorms NXT server: motors, sensors and other devices. There is one type of motor called **Servo**. There are four types of sensors: **Switch**, **SoundSensor**, **UltraSonicSensor** and **LightSensor**. Two other devices are available: **Battery** and **Beep**. A complete description of these devices is available in [Section 29.5](#). Writing a layout entails instantiating particular objects from the devices above.

29.4.1 Instantiating Motors

The Servo constructor has the following syntax:

```
Servo.new(port);
```

urbiscript
Session

where *port* is either "A", "B" or "C". The created object makes it possible to control the motor connected to the port in parameter. As an example, in the current 'TriBot.u', three Servo objects are created:

```
wheelL = Servo.new("C");
wheelR = Servo.new("A");
claw = Servo.new("B");
```

urbiscript
Session

The wheel left is on port C, the wheel right on A and the claw is on port B.

29.4.2 Instantiating sensors

To create a Switch object, the syntax is the following:

urbiscript
Session

```
Switch.new(port);
```

where *port* can be 1, 2, 3 or 4.

In ‘TriBot.u’ the switch is used as a bumper plugged on the port 4 so the instantiation is:

urbiscript
Session

```
bumper = Switch.new(4);
```

UltraSonicSensor objects are created the same way:

urbiscript
Session

```
UltraSonicSensor.new(port);
```

port can be 1, 2, 3 or 4.

In ‘TriBot.u’ an ultrasonic sensor is connected on port 2:

urbiscript
Session

```
sonar = UltraSonicSensor.new(2);
```

The SoundSensor object has an additional parameter:

urbiscript
Session

```
SoundSensor.new(port, mode);
```

port can be 1, 2, 3 or 4. *mode* sets the mode of the sensor. It can be “DB” or “DBA”, meaning “decibel” or “decibel adjusted” (the latter records sounds in the frequency range 200-14000Hz).

In ‘TriBot.u’, the sound sensor is on port 1 using decibel adjusted:

urbiscript
Session

```
decibel = SoundSensor.new(1, "DBA");
```

LightSensor also has two parameters:

urbiscript
Session

```
LightSensor.new(port, mode);
```

port can be 1, 2, 3 or 4. *mode* sets the sensor mode and it can be “Ambient” (so the sensor measures the ambient light), “Reflector” (the sensor lights on its led and measures the reflected light) and “Normal” (the sensor returns its raw value).

In ‘TriBot.u’, the light sensor is in reflector mode on port 1:

urbiscript
Session

```
light = LightSensor.new(3, "Reflector");
```

29.4.3 Other devices

The Battery device makes it possible to get the current battery level.

urbiscript
Session

```
Battery.new;
```

In ‘TriBot.u’:

urbiscript
Session

```
battery = Battery.new();
```

The Beeper device makes it possible to emit customized beeps from the NXT speaker.

urbiscript
Session

```
Beeper.new();
```

In ‘TriBot.u’:

urbiscript
Session

```
beeper = Beeper.new();
```

29.5 Available UObject Devices

This section exhaustively describes the UObjects available in the LEGO Mindstorms NXT Urbi server.

29.5.1 Servo

Servo describes a motor. A Servo instance contains the following slots:

- **init(*port*)**

The UObject constructor. *port* is a port name. (See previous slot *port*). Example:

```
wheel = Servo.new("A");
```

urbiscript
Session

- **port**

The port where the servo is plugged. It can be "A", "B" or "C". You can change the value while the server is running. Beware, if you change the port, that will free the old port (so you will be able to create another device on it) and will busy the new one (so you won't be able to create another device on it).

- **speed**

Motor's speed, from -100 to 100.

- **val**

Motor's position.

29.5.2 UltraSonicSensor

UltraSonicSensor describes a sonar sensor. An UltraSonicSensor instance contains the following slots:

- **init(*port*)**

The UObject constructor. *port* is a port name. (See previous slot *port*). Example:

```
sonar = UltraSonicSensor.new(1);
```

urbiscript
Session

- **port**

The port where the servo is plugged. It can be 1, 2, 3 or 4. You can change the value while the server is running. Beware, if you change the port, that will free the old port (so you will be able to create another device on it) and will busy the new one (so you won't be able to create another device on it).

- **val**

Distance in cm (from 0 to 255).

29.5.3 SoundSensor

SoundSensor describes a sound sensor. A SoundSensor instance contains the following slots:

- **init(*port*, *mode*)**

The UObject constructor. *port* is a port name. (See previous slot *port*). *mode* is a mode name. (See previous slot *mode*). Example:

```
decibel = SoundSensor.new(1, "DB");
```

urbiscript
Session

- **mode**

"DB" is decibel or "DBA" is decibel audible (measures only in the audible frequencies range).

- **port**

The port where the servo is plugged. It can be 1, 2, 3 or 4. You can change the value while the server is running. Beware, if you change the port, that will free the old port (so you will be able to create another device on it) and will busy the new one (so you won't be able to create another device on it).

- **val**
The sound level measured (from 0 to 1).

29.5.4 LightSensor

LightSensor describes a light sensor. A LightSensor instance contains the following slots:

- **init(*port*, *mode*)**
The UObject constructor. *port* is a port name. (See previous slot *port*). *mode* is a mode name. (See previous slot *mode*). Example:

```
urbscript
Session
light = LightSensor.new(1, "Ambient");
```

- **mode**
 - "Ambient" measures the ambient light.
 - "Reflector" lights on a led and measures the reflection.
 - "Normal" returns the raw value
- **port**
The port where the servo is plugged. It can be 1, 2, 3 or 4. You can change the value while the server is running. Beware, if you change the port, that will free the old port (so you will be able to create another device on it) and will busy the new one (so you won't be able to create another device on it).
- **val**
The light level measured (from 0 to 1).

29.5.5 Switch

Switch describes a touch sensor. A Switch instance contains the following slots:

- **init(*port*)**
The UObject constructor. *port* is a port name. (See previous slot *port*). Example:

```
urbscript
Session
bumper = Switch.new(1);
```

- **port**
The port where the servo is plugged. It can be 1, 2, 3 or 4. You can change the value while the server is running. Beware, if you change the port, that will free the old port (so you will be able to create another device on it) and will busy the new one (so you won't be able to create another device on it).
- **val**
The switch status (0 or 1).

29.5.6 Battery

Battery describes a battery device. A Battery instance contains the following slots:

- **init**
The UObject constructor. Example:

```
urbscript
Session
battery = Battery.new;
```

- **val**
Battery remaining power between 0 and 1.

29.5.7 Beeper

Beeper describes a speaker device. A Beeper instance contains the following methods:

- **init**

The UObject constructor. Example:

```
beeper = Beeper.new;
```

urbiscript
Session

- **play(*frequency*, *duration*)**

Plays a beep of a custom frequency for a custom duration. *frequency* is the frequency of the beep (in Hz), between 200 and 14000. *duration* is the duration of the beep (in seconds), 0 is infinite. Example:

```
beeper.play(200,1s);
```

urbiscript
Session

29.5.8 Command

Command allows you to use expert commands. The Command UObject contains the following methods:

- **request(*bufferIn*, *sizeOut*)**

Send a command requiring an answer, and return the buffer out. This is only recommended to expert users. Beware of the *sizeOut* parameter. Indeed if it is too long, the request will normally fail and when the size is too short, the command may work but all Mindstorms internal buffers will be shift back (so this will bend all other requests).

bufferIn is the buffer of the command. *sizeOut* is the size of the answer.

Example:

```
answer = Command.request([7,0], 15);
```

urbiscript
Session

that returns the information on input port 0.

- **send(*buffer*)**

Sends a command without requiring any answer. *buffer* must be a list of integers between 0 and 255. Example:

```
Command.send([3,10,10,0,0]);
```

urbiscript
Session

that plays a beep.

29.5.9 Instances

Here are the different instances in the ‘TriBot.u’ layout.

Instance	UObject	Description
wheelL	Servo	Left wheel
wheelR	Servo	Right wheel
claw	Servo	Claw
sonar	UltraSoundSensor	Distance sensor
decibel	SoundSensor	Sound sensor
light	LightSensor	Light sensor
bumper	Switch	Touch sensor
beeper	Beeper	Emits beeps
battery	Battery	Battery

29.5.10 Groups

Here are the different groups provided in the current layout.

Group Name	Description
wheels	The two wheels
motors	All the motors
sensors	All the sensors
hardware	All the devices

Chapter 30

Gostai Open Jazz



Disclaimer

The information contained in this chapter is subject to change without notice. Gostai makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Gostai shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damage in connection with the furnishing, performance, or use of this material.

This chapter contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws. Use of this manual is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in its present form or with alterations, is expressly prohibited.

30.1 Getting started

30.1.1 Documentation and Support

30.1.1.1 Jazz User Manual

Jazz User Manual contains all the information needed to set up your robot, and daily use it: control it with its joystick, control it remotely through Internet (Jazz Connect), set up watch for your premises and configure alerts (Jazz Security), or launch special behaviors (Jazz Connect).

30.1.1.2 Getting Support

Gostai provides commercial support and services, in the form of support packs that you can purchase (http://gostai.com/support/support_packs/). Gostai experts can help you master the Jazz SDK, and assist you with your Jazz development. To contact Gostai support: support@gostai.com.

30.1.2 Requirements

In order to communicate with, and program Jazz, you need

- a Local Area Network (LAN) with DHCP.
- a WiFi access point to your LAN. ESSID **must not** be hidden.
- a computer connected to your LAN with at least
 - 1.5 GHz CPU
 - 512 MB RAM
 - WiFi card (for wireless connection between computer and Jazz in ad-hoc mode)
 - Mac OS X Snow Leopard 10.6, GNU/Linux 32 bits, Windows XP 32 bits
 - Urbi SDK 2.7.3
 - ssh client (putty on Windows, ssh on GNU/Linux or Mac OS X)
 - scp client (pscp or WinSCP on windows, scp on GNU/Linux or Mac OS X)
 - telnet client (Gostai Lab, Gostai Console, GNU Netcat)

30.1.3 Embedded Software

- Ubuntu GNU/linux 10.04 server
- Gostai Urbi 2.7.3 runtime and SDK

30.1.4 Connecting Jazz to your Local Area Network

Please refer to Jazz user manual to know how to setup Jazz on your LAN.

30.1.5 Connecting to Jazz from your Computer

30.1.5.1 Obtaining Jazz IP Address

Turn on your robot with the power button. If you have correctly configured your robot to connect on your LAN (see Jazz user manual), it will say its IP address upon start-up. You will use this IP address to access Jazz on your LAN.

If Jazz cannot access LAN, it will start in ad-hoc mode and won't say its IP address. Please refer to Jazz user manual to know how to setup Jazz on your LAN.

If DNS is activated on your Local Area Network, then you can also access Jazz using his host name instead of its IP address. In this documentation A1111FRPA001001 is the Jazz host name.

30.1.5.2 Gostai Suite

Gostai Studio Please refer to http://gostai.com/products/studio/gostai_studio/ to get more information about Gostai Studio.

Gostai Lab Please refer to http://gostai.com/products/studio/gostai_lab/ to get more information about Gostai Lab.

30.1.5.3 Telnet Connection to Urbi on Jazz

Jazz runs the Urbi software platform which exposes all the interfaces you need to program your Jazz robot. From a separate computer, you can connect through the network to the Urbi runtime running in the robot and directly send commands written in urbiscript.

By default, when your Jazz robot is powered on, the Urbi runtime listen to TCP connections established on port 54000. You have to use a telnet client to establish such a connection.

From a GNU/Linux or Mac OS X computer, we recommend that you use netcat (nc) in combination with the rlwrap tool (add nice editing capabilities to your netcat connection, keep a commands history).

```
$ rlwrap nc A1111FRPA001001 54000
[00732235] *** ****
[00732235] *** Jazz version 2.10.2 rev. a2aa56e
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** Urbi SDK version 2.7.2 rev. 7a6a48f
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00732235] *** be used under certain conditions. Type 'license;', 
[00732235] *** 'authors;', or 'copyright;' for more information.
[00732235] ***
[00732235] *** Check our community site: http://www.urbiforge.org.
[00732235] *** ****
21+21;      // Then type urbiscript commands
[10421607] 42 // Urbi runtime sends you returns value if available
```

Shell Session

From a Windows computer, we recommend that you use Gostai Console (http://www.gostai.com/download/gostai_console/, see Figure 30.1), a dedicated tool to connect to the Urbi engine running on your Jazz robot.

Specify the host name and port to use ('A1111FRPA001001:54000') in the text field in the top of the window and click on the right to start the connection.

30.1.5.4 Ssh Access to Jazz Filesystem

You can use ssh to connect to Jazz and access its filesystem. If you are using Windows you have to install Putty (<http://www.putty.org>). On Mac OS X or GNU/Linux use the ssh built-in client. To connect through ssh you have to specify in your ssh client:

- Jazz IP address (obtained at robot start up) or host name
- default user login: 'gostai' (do not copy the quotes)
- default user password: 'gostai' (do not copy the quotes)

For example, if Jazz IP address on your LAN is '192.168.1.3', then under GNU/Linux you would have to type in a shell:

```
$ gostai@192.168.1.3
Login in
gostai@A1111FRPA001001:~$ # Now you are logged in Jazz, you can type shell commands
```

Shell Session

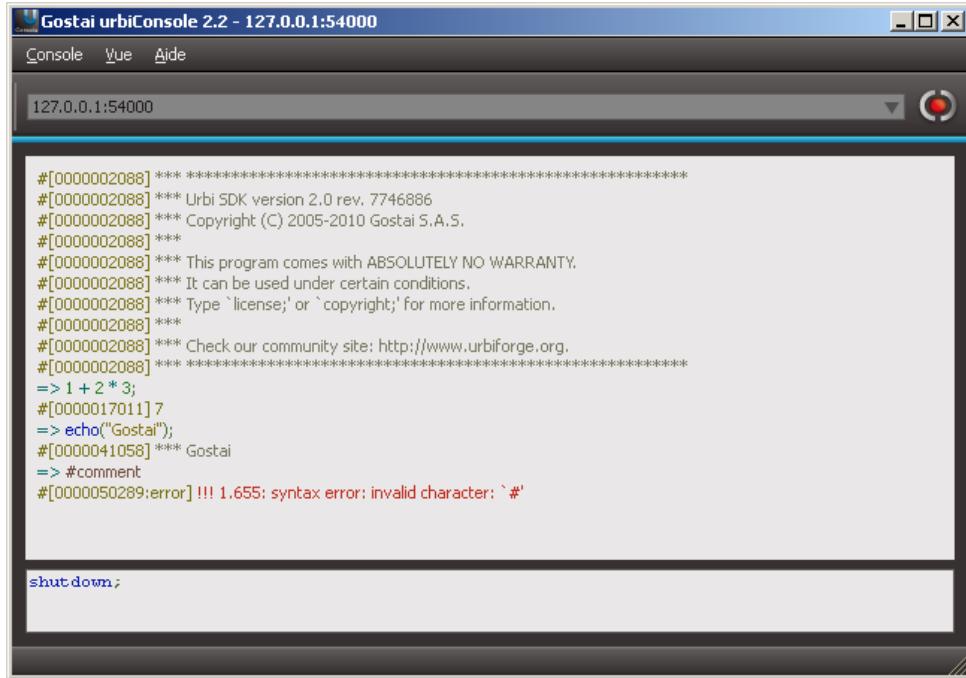


Figure 30.1: A Gostai Console session.

If Jazz host name is ‘A1111FRPA001001’ and you have DNS resolution activated in your LAN, type:

```
Shell
Session
$ ssh gostai@A1111FRPA001001
Login in
gostai@A1111FRPA001001:~$ # Now you are logged in Jazz, you can type shell commands
```

Once you are connected with ssh to your Jazz robot, you arrive in a Bash shell which gives you complete access to Jazz filesystem. If you are not familiar with GNU/Linux and the Bash shell, we advise that you read some Bash tutorial on the Internet to familiarize with Bash basic commands.

It is not mandatory to use ssh to program your Jazz robot. It will only becomes useful if you want to use advanced features like embedding your code in Jazz and have it load at robot start up.

30.1.6 Jazz internal website

Jazz embed a website that allows you to configure and control it. To access this website, just enter Jazz IP address in your browser (Internet explorer, Firefox, Chrome, ...).

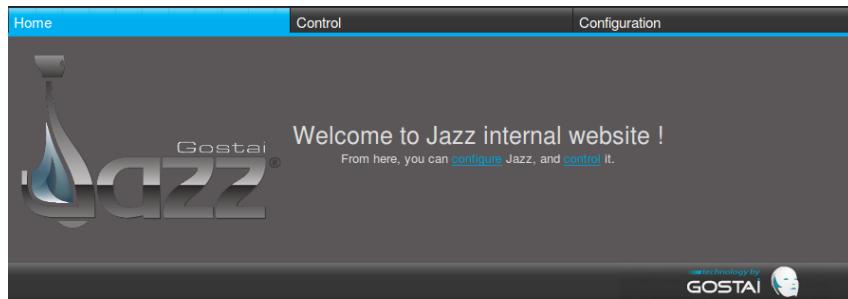


Figure 30.2: Jazz Internal Website.

Click on the *Control* menu category to access Jazz local control interface, or click on the *Configuration* menu to access Jazz configuration pages (WiFi, services launched, ...).

30.1.7 Uploading and loading your code on Jazz

30.1.7.1 The *User Services* web page

In Jazz internal website ([Section 30.1.6](#)), we provide a page you can use to upload your own urbiscript services to Jazz, and made it execute them at startup. To access this page, click on the *Configuration* menu, then click on the *User services* sub-menu.

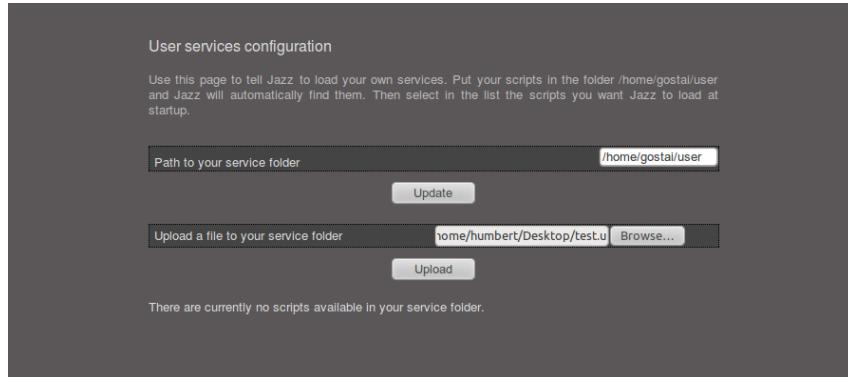


Figure 30.3: User services web page.

From this page you can change the folder where Jazz will search for your code (by default it's `/home/gostai/user`). To change it, edit the text field after the label *Path to your service folder*, then press the *Update* button below.

You can also upload urbiscript source file to this folder. Click on the *Browse* button next to the *Upload a file to your service folder* label. Then choose a file on your computer. It will be automatically uploaded to the robot service folder when you press the *Upload* button. Once your file was uploaded, it will be listed on the web page.

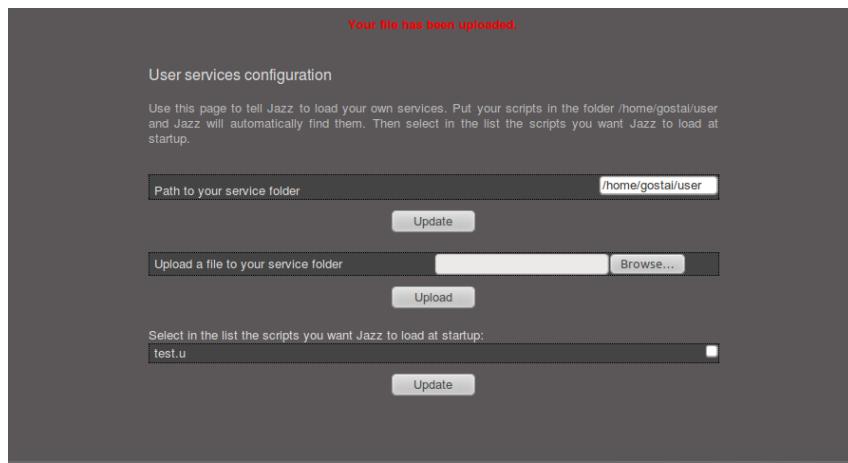


Figure 30.4: User services web page: *test.u* service was uploaded.

If you want Jazz to automatically load your urbiscript file at startup, check the check-box next to your file name, and click on the *Update* button.

30.1.7.2 Uploading Files on Jazz Filesystem using scp

In order to upload files on Jazz filesystem you need to have a client supporting the `scp` protocol.

Under GNU/Linux or similar Unix system, you can use the `scp` command line tool.

```
$ # Copy 'myfile.u' to Jazz home folder:  
$ scp myfile.u gostai@A1111FRPA001001:/home/gostai/myfile.u
```

Shell
Session

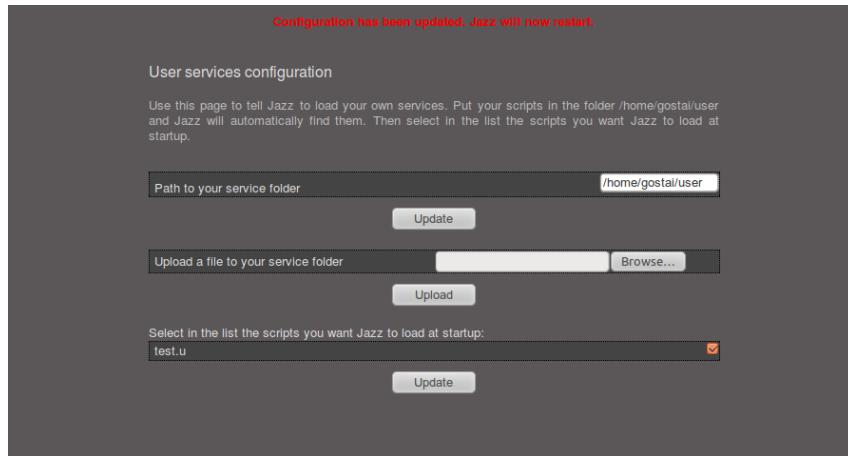


Figure 30.5: User services web page: *test.u* service loaded at startup.

Under windows you can use WinSCP (<http://winscp.net/eng/docs/lang:fr>) or Pscp, the scp client that comes with Putty (<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>).

You can then verify with ssh that your file is in the robot:

Shell Session

```
$ ssh gostai@A1111FRPA001001
Login in
gostai@A1111FRPA001001:~$ ls /home/gostai/myfile.u
/home/gostai/myfile.u
```

30.1.8 Updating Jazz Software

Gostai provides updates to Jazz software. To verify if updates are available and automatically install them, go to Jazz internal website (Section 30.1.6), click on the *Configuration* menu, then click on the *System* sub-menu.

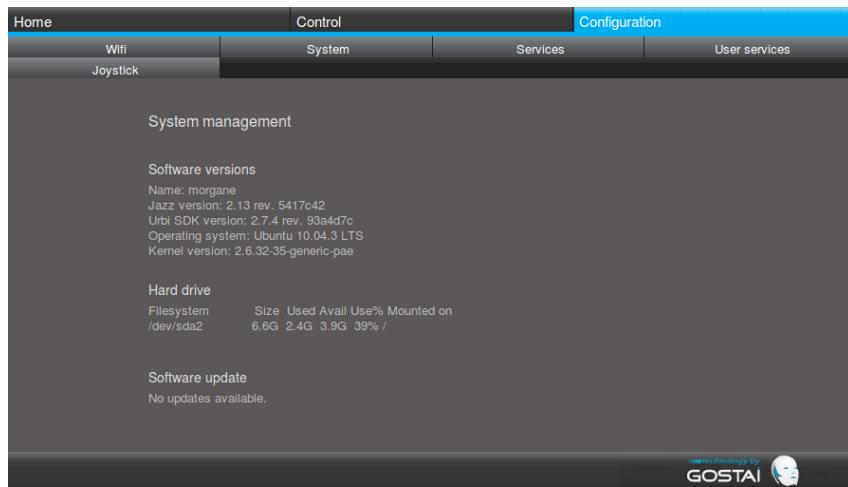


Figure 30.6: System page: display Jazz software information and allow updates.

Jazz will check if updates are available, and in case there are, it will display a button that allows you to start the update process. Please do not shutdown Jazz while updating. At the end of the update Jazz will display a message in its website and automatically restart.

If no updates are available, the message "No updates available" will be displayed instead of the update button.

30.1.9 Restarting Jazz Software

To start, stop or restart Jazz software (the Urbi runtime plus Jazz Urbi software), you can use the script `/etc/init.d/jazz`:

```
gostai@A1111FRPA001001:~$ # shutdown Jazz
gostai@A1111FRPA001001:~$ /etc/init.d/jazz stop
gostai@A1111FRPA001001:~$ # start Jazz
gostai@A1111FRPA001001:~$ /etc/init.d/jazz start
gostai@A1111FRPA001001:~$ # restart Jazz
gostai@A1111FRPA001001:~$ /etc/init.d/jazz restart
```

Shell
Session

30.1.10 Changing Software Profile

The functionality described here is an advanced functionality. Do not use it without first asking advises to Gostai. Misuse can result in Jazz instability or breakage.

Gostai Jazz robot software is modular, and can operate with different software profiles in function of how you want to use the robot: you can activate all its functionalities, or only part of them. We provide an interface that allow to fine tune Jazz in function of your use cases.

To access this interface go to Jazz internal web interface ([Section 30.1.6](#)), and choose the *Configuration* menu, then click on the *Services* sub-menu.

You can use the shortcut button to set Jazz predefined default configuration, or directly fine tune Jazz activating only the functionalities you need. Please be cautious when doing so, some services have dependencies between each other, some combinations of services might result in Jazz instability. Always request advises from Gostai.

Press the send button when your are done to save the new configuration. Jazz will automatically restart it's software using the new configuration.

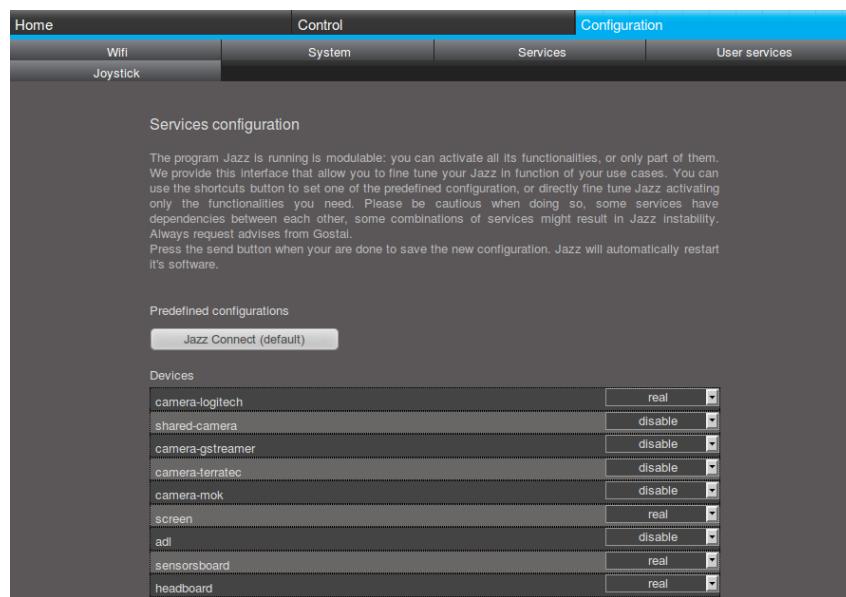


Figure 30.7: Service activation page: tune Jazz software profile.

30.1.11 Using the Examples

With the descriptions of Jazz modules API, we include urbiscript as well as UObjects examples to help you understand how to use the modules.

30.1.11.1 urbiscript Examples

To play with urbiscript on Jazz, you can open a telnet connection to your robot (use netcat or Gostai Console), and copy and paste the example to evaluate the code.

Shell Session

```
$ rlwrap nc A1111FRPA001001 54000
[00732235] *** ****
[00732235] *** Jazz version 2.10.2 rev. a2aa56e
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** Urbi SDK version 2.7.2 rev. 7a6a48f
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00732235] *** be used under certain conditions. Type 'license;',
[00732235] *** 'authors;', or 'copyright;' for more information.
[00732235] ***
[00732235] *** Check our community site: http://www.urbiforge.org.
[00732235] *** ****
echo("hello world");
[00138887] *** hello world
```

Alternatively, you can also write the urbiscript code to a file, upload this file on Jazz filesystem using `scp`, and then load the code from a telnet connection.

Shell Session

```
$ cat test.u
echo("hello world");
$ scp test.u gostai@A1111FRPA001001:/home/gostai/test.u
$ rlwrap nc A1111FRPA001001 54000
[00732235] *** ****
[00732235] *** Jazz version 2.10.2 rev. a2aa56e
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** Urbi SDK version 2.7.2 rev. 7a6a48f
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00732235] *** be used under certain conditions. Type 'license;',
[00732235] *** 'authors;', or 'copyright;' for more information.
[00732235] ***
[00732235] *** Check our community site: http://www.urbiforge.org.
[00732235] *** ****
load("/home/gostai/test.u");
[00138887] *** hello world
```

30.1.11.2 UObjects examples

See the Urbi SDK documentation to know how to compile UObjects with Urbi SDK (using either `umake-shared` ([Section 22.10.2](#)) or Visual Studio templates).

Once you have compiled your UObject, you can run it remotely from your computer, or embed it in the robot (you have to compile your UObject with GNU/Linux for it to be usable on the robot).

Compiling on Jazz Compiling on Jazz can be very slow and inconvenient, however it frees you from the need of installing the Urbi SDK on your computer. To compile on Jazz, first verify that `umake-shared` is available in your PATH.

Shell Session

```
gostai@A1111FRPA001001:~$ which umake-shared
/usr/local/stow/urbi-sdk/bin//umake-shared
```

In case `umake-shared` is not found, update the PATH variable.

Shell Session

```
gostai@A1111FRPA001001:~$ which umake-shared
gostai@A1111FRPA001001:~$ export PATH=/usr/local/stow/urbi-sdk/bin:$PATH
gostai@A1111FRPA001001:~$ which umake-shared
/usr/local/stow/urbi-sdk/bin//umake-shared
```

Then stop Jazz runtime to free the processor for your compilation.

```
gostai@A1111FRPA001001:~$ /etc/init.d/jazz stop
```

Shell Session

Upload some UObject code to your robot using `scp`.

```
gostai@A1111FRPA001001:~$ cd myuob
gostai@A1111FRPA001001:~/myuob$ ls
test.cc
gostai@A1111FRPA001001:~/myuob$ cat test.cc
#include <urbi/uobject.hh>

class Test : public urbi::UObject
{
public:
    Test(const std::string& s)
        : UObject(s)
    {
        UBindFunction(Test, init);
    }

    int init()
    {
        UBindFunction(Test, foo);
        return 0;
    }

    std::string foo()
    {
        return "Hello world";
    }
};

UStart(Test);
```

Shell Session

Now you can compile your UObject.

```
gostai@A1111FRPA001001:~/myuob$ umake-shared -q -o test
gostai@A1111FRPA001001:~/myuob$ ls test.so
test.so
```

Shell Session

You obtain a shared library containing your UObject binary code, that you can load and use from urbiscript.

```
loadLibrary("/home/gostai/myuob/test.so");
Test;
[00711192] Test
var t = Test.new;
[00718330] Test_0xfffffffffb33ab028
t.foo;
[00713182] "Hello world"
```

urbiscript Session

Compiling on your Computer See the Urbi SDK documentation to compile on your computer. Use the same version of Urbi SDK that the one used in Jazz for compatibility. To get the version of Urbi used in Jazz see [Section 30.5.1](#).

30.2 urbiscript API

30.2.1 Eye leds

You can change the color of the leds located in the eyes.

30.2.1.1 robot.body.head.eye

- **b**
Blue component (0-255).
- **g**
Green component (0-255).
- **r**
Red component (0-255).

The eyes led objects are available using the Gostai Standard Robotics API compliant name `robot.body.head.eye[left]` and `robot.body.head.eye[right]`.

30.2.1.2 Example

urbiscript
Session

```
// Set the color of the leds in Jazz eyes to green
function green_eyes()
{
    do (robot.body.head)
    {
        eye[left].r = 0 & eye[left].g = 255 & eye[left].b = 0 &
        eye[right].r = 0 & eye[right].g = 255 & eye[right].b = 0
    };
}};

// Call the function: Jazz eyes becomes green
green_eyes;
```

30.2.2 Docking

Jazz robot have automatic docking to its charging station capabilities. The function to launch the algorithm is, of course, Gostai Standard Robotics API compliant. For a description of Jazz docking API please refer to [??.](#)

30.2.3 Head Motors

Jazz's head has two degrees of liberty: horizontal (`jazz.body.head.yaw`) and vertical (`jazz.body.head.pitch`)

30.2.3.1 robot.body.head

- **pitch**
Vertical position of Jazz's head. You can read or set the position using the `val` attribute.
- **yaw**
Horizontal position of Jazz's head.

30.2.3.2 Example

```
// Makes Jazz head oscillate horizontally, as if it was saying "no".
// The function stops after the specified duration.
function sayNo(duration)
{
    timeout (duration) {
        // Here we assign a sinusoidal trajectory to the head yaw motor:
        robot.body.head.yaw.val = sin:2s ampli:0.7;
    };
};

// Make Jazz say "no" for 10 seconds.
sayNo(10s);
```

urbiscript
Session

30.2.4 Microphone

30.2.4.1 Quickly Record the Audio Stream

To quickly record your robot audio microphone stream from your computer, you can use the command line tool `urbi-sound` ([Section 22.9](#)).

For example to record 20 seconds of audio, ssh on the robot and do:

```
gostai@A1111FRPA001001:~$ urbi-sound -h
usage: urbi-sound [options]
      record and play (or save) sound from a robot

Options:
  -h, --help                  display this message and exit successfully
      --version                display version information
  -H, --host=HOST              address to connect to
  -P, --port=PORT              port to connect to
      --port-file=FILE          read port number in FILE
  -d, --device=DEVICE          query sound on DEVICE.val (default: micro)
  -D, --duration=DURATION     recording duration in seconds
  -o, --output=FILE            save sound in FILE (/dev/dsp)
  -n, --no-headers             do not include the headers when saving the sound
gostai@A1111FRPA001001:~$ urbi-sound -H localhost -D 20 -o sample.wav
gostai@A1111FRPA001001:~$ aplay sample.wav
```

Shell
Session

30.2.4.2 `robot.body.head.micro`

The microphone device has the following slots:

- **aec**
EchoCanceller ([uobjects.EchoCanceller](#)) object.
- **agc**
AutomaticGainControl ([Interface.AutomaticGainControl](#)) interface
- **alsa**
AlsaMicrophone ([uobjects.AlsaMicrophone](#)) instance.
- **denoise**
Denoiser ([Interface.Denoiser](#)) interface
- **gainPercent**
Microphone gain amplification, expressed in percent (0 to 100).
- **load**
Enable or disable the microphone.

- **val**

Binary value corresponding to the sound captured by the microphone, and processed by AutomaticGainControl and Denoise algorithms (if they are enabled).

The microphone object is available using the Gostai Standard Robotics API compliant name `robot.body.head.micro` or directly typing the shortcut name `micro`.

30.2.4.3 Example

urbiscript
Session

```
// Calculate the microphone refresh rate by counting the number of time the
// microphone value is refreshed during 5 seconds, then dividing this count
// by 5.
var count = 0 |
var lnk = micro.&val.notifyChange(closure (){count++}) |
sleep(5s) |
micro.&val.removeNotifyChange(lnk) |
var refresh_rate = count / 5 |
// Retrieve sound information in the value header
var keys = micro.val.keywords.split(" ") |
// Display all the information
echo("Micro information:");
echo(" format: %s" % keys[0]);
echo(" sample format: %s" % keys[1]);
echo(" rate: %s" % keys[2]);
echo(" sample size: %s" % keys[3]);
echo(" channels: %s" % keys[4]);
echo(" micro.val is refreshed %s times per second" % refresh_rate);
[19281072] *** Micro information:
[19281073] *** format: raw
[19281074] *** sample format: 1
[19281076] *** rate: 16000
[19281077] *** sample size: 16
[19281080] *** channels: 1
[19281086] *** micro.val is refreshed 25 times per second
```

urbiscript
Session

```
// Let's record 5 seconds of the micro sound:
// - create the file where we'll save the sound, and open it in an output stream
var o = OutputStream.new(File.create("/tmp/sound.pcm"));
// - every time the micro value change, append the sound chunk to the file
var lnk = micro.&val.notifyChange(closure() { o << micro.val.data ; });
sleep(5s);
// - stop recording
micro.&val.removeNotifyChange(lnk);
// - close the output stream
o.close;
// Now you can play the recorded sound with the shell command:
//   aplay -f S16_LE -r 16000 -c 1 -t raw /tmp/sound.pcm
```

30.2.5 Network

Network object provides methods and attributes to configure and get the current state of the WiFi.

30.2.5.1 robot.network

- `dnsAddress`

- `essid`

- gatewayAddress
- ipAddress
- key
- netMask
- submit

30.2.5.2 Example

To connect your Jazz robot to your LAN, use the web interface as describe in the Jazz User Manual. It is also possible to specify the network on which Jazz connects. However be careful, Jazz save the network information you provide, and will reuse them for all its future connections. To connect to a WiFi network with ESSID *myNetwork* and key *myHiddenKey*.

```
network.essid="myNetwork";
network.key="myHiddenKey";
// Tell network to connect with current parameters and save them.
network.submit;
```

urbiscript
Session

You can also provide a static IP. Let say we want IP 192.168.0.3, net-mask 255.255.255.0, DNS and gateway 192.168.0.1. Using `do` to factor `network`:

```
do (network)
{
    essid="myNetwork";
    key="myHiddenKey";
    ipAddress="192.168.0.3";
    netMask="255.255.255.0";
    dnsAddress="192.168.0.1";
    gatewayAddress="192.168.0.1";
    // Tell network to connect with current parameters and save them.
    submit;
};
```

urbiscript
Session

To reset the robot in ad-hoc mode:

```
// The robot will take IP 192.168.0.1 and start a DHCP server.
network.setAdHoc;
```

urbiscript
Session

30.2.6 Playing Sound

Jazz can play mp3 files with its speaker. The files must be available on Jazz filesystem, or hosted on a web server.

30.2.6.1 Example

```
mp3.play("/usr/local/stow/jazz/share/mp3/jazz_startup.mp3");
sleep(1s);
mp3.stop;
mp3.play("http://example.com/sound/sound.mp3");
```

urbiscript
Session

30.2.7 Text to Speech

Jazz integrate text to speech. It must be connected to Internet to use text to speech). Only french language is currently supported.

30.2.7.1 Example

For Jazz to say the sentence: “Bonjour, je suis Jazz le robot”, run:

urbiscript
Session

```
voice.say("Bonjour, je suis Jazz le robot");
```

Note: the function can fail in case Jazz network connection timeout.

30.2.8 Video Camera

You have access to Jazz video camera device: you can request the raw images, and process them using urbiscript/UObject.

30.2.8.1 Visualize the Video Stream

To quickly visualize your robot video camera stream from your computer, you can use Gostai Lab or the command line tool `urbi-image` ([Section 22.4](#)).

`urbi-image` opens a graphic windows rendering the live stream from the video camera. Its usage is:

Shell
Session

```
$ urbi-image -h
usage: urbi-image [options]
Display images from an urbi server, or save one image if -o is given

Options:
-h, --help           display this message and exit successfully
--version          display version information
-H, --host=HOST    address to connect to
-P, --port=PORT    port to connect to
--port-file=FILE   read port number in FILE
-p, --period=PERIOD query images at given period (in milliseconds)
-F, --format=FORMAT select format of the image (rgb, ycrcb, jpeg, ppm)
-r, --reconstruct  use reconstruct mode (for aibo)
-j, --jpeg=FACTOR  jpeg compression factor (from 0 to 100, def 70)
-d, --device=DEVICE query image on DEVICE.val (default: camera)
-o, --output=FILE   query and save one image to FILE
-R, --resolution=RESOLUTION select resolution of the image (0=bigest)
-s, --scale=FACTOR  rescale image with given FACTOR (display only)

transfer Format : jpeg=transfer jpeg, raw=transfer raw
save    Format : rgb , ycrcb, jpeg, ppm
$ urbi-image -H A1111FRPA001001
```

To visualize the stream with Gostai Lab, connect Gostai Lab to your robot, and enter `camera.val` in the variable field. The live stream of the video camera will automatically get displayed.

30.2.8.2 robot.camera

The camera device is represented by the urbiscript object whose slots are:

- **height**
Video camera images height. (const)
- **val**
Current acquired image (in RGB format).

- **width**

Video camera images width. (const)

The camera object is available using the Gostai Standard Robotics API compliant name `robot.body.head.camera` or directly typing the shortcut name `camera`.

30.2.8.3 Example

Here some urbiscript example code manipulating the camera object:

```
echo("Acquired image size: %sx%s" % [camera.width, camera.height]);
[00237095] *** Acquired image size: 320x240
var e = watch(camera.val);
var t = Timeout.new(5s);
try
{
    var count = 0 |
    t:at(e?) count++ &
    t:every(1s) {
        echo("camera.val was refreshed %s times" % count);
        count = 0;
    };
}
catch (var e)
{
    echo("timed out");
};
[00238639] Event_Oxfffffff4dc3728
[00238641] Timeout_Oxfffffff4dd8928
[00238643] *** camera.val was refreshed 0 times
[00239647] *** camera.val was refreshed 14 times
[00240647] *** camera.val was refreshed 15 times
[00241647] *** camera.val was refreshed 15 times
[00242647] *** camera.val was refreshed 15 times
[00243647] *** camera.val was refreshed 15 times
[00243650] *** timed out
```

urbiscript
Session

If you want to process the video camera raw image data, you can only do it from C++ or Java, using the UObject component API.

```
#include <urbi/uobject.hh>

// ProcessCamera is an UObject that does video processing with the
// camera stream.
class ProcessCamera : public urbi::UObject
{
public:
    ProcessCamera(const std::string& s)
        : urbi::UObject(s)
    {
        UNotifyChange("camera.val", &ProcessCamera::process);
    }

    // Each time camera.val urbiscript variable is refreshed, your
    // function will get called with the new video camera image.
    void process(urbi::UVar& val)
    {
        urbi::UIImage i = val;
        // Do processing. See Urbi SDK documentation to know more about UIImage.
    }
};
```

C++

30.2.9 Movement

We provide high level API to move Jazz.

30.2.9.1 urbiscript interface

The following slots of the `robot` object control the movement.

- `aborted?(commandName, reason)`

Event emitted when a command (`go`, `turn`, `goTo`, `goToAbsolute`, `turnAbsolute`, `goToChargingDock`) is aborted (generally because another command has started). `commandName` is the string name of the command. `reason` gives the reason of the abortion.

- `finished?(commandName)`

Event emitted when a command (`go`, `turn`, `goTo`, `goToAbsolute`, `turnAbsolute`, `goToChargingDock`) finishes successfully. `commandName` is the string name of the command.

- `go(distance)`

Go forward if `distance` > 0, go backward if `distance` < 0. Distance is in meter.

- `goTo(x, y, [z], [yaw])`

Go to `x`, `y` in meter. `x` > 0 corresponds to the front of robot. `y` > 0 corresponds to the left of robot. `z` is optional and unused. `yaw` is optional and is the angle the robot must reach at the end of the motion (`yaw` = 0 is the direction when the robot receives the command).

- `goToAbsolute(x, y, [z], [yaw])`

Go to a position in the absolute initial frame (`x` = 0, `y` = 0 and `yaw` = 0 is the initial position).

- `goToChargingDock`

Go to charging dock if available.

- `leaveChargingDock`

Make the robot leave the charging dock. This is the only command allowed if the robot is charging. If not, it has no effects.

- `moving`

Whether the robot is moving.

- `position`

List [x, y, z, yaw] given the absolute position of the robot in the absolute initial frame `x` > 0 corresponds to the front of the robot at initial position `y` > 0 corresponds to the left of the robot at initial position `yaw` = 0 corresponds to the direction of the robot at initial position.

- `renunciationInterval`

If ≤ 0 , renunciation is not used. If > 0 , any command stops (the event `unreachable` is emitted) if no motion are observed during at least the given value (in seconds).

- `started?(commandName)`

Event emitted when a command (`go`, `turn`, `goTo`, `goToAbsolute`, `turnAbsolute`, `goToChargingDock`) is sent. `commandName` is the string name of the command.

- `stop`

Stop all moves.

- `turn(angle)`

Turn left if `angle` > 0, turn right if `angle` < 0. `angle` is in radian.

- **turnAbsolute(yaw)**

Turn toward a direction in the absolute initial frame ($yaw = 0$ is the initial direction).

- **watchdogInterval**

If ≤ 0 , watchdog is not used. If > 0 , time left for any command to finish before it is aborted by the watchdog.

- **xSpeed**

Linear speed.

- **yawSpeed**

Angular speed (> 0 if the robot turns left).

30.2.9.2 Example

```
var watchdogGoTag = Tag.new();

function goDuring(duration, speed)
{
    disown({
        watchdogGoTag.stop |
        watchdogGoTag:{
            robot.xSpeed = speed;
            sleep(duration);
            robot.stop();
        };
    });
    true;
}();

goDuring(2s, 1);
robot.go(2s);
```

urbiscript
Session

30.2.10 Screen Display

The Open Jazz does not provide yet any Urbi API to manage the display. However you can launch GNU/Linux graphical commands which will use the display. An X server (Xorg see <http://www.x.org/wiki/>) is installed in Jazz to manage the display.

30.2.10.1 From urbiscript

You can launch any GNU/Linux application from urbiscript by using the **Process** object. You can try to launch the **xeyes** application to test the display. To do so, connect to the Urbi runtime and create an **xeyes** process, then run it.

```
var p = Process.new("xeyes", ["-geometry", "800x500"]);
[00059674] Process xeyes
p.run;
// Two eyes get displayed on the screen.
sleep(10s);
p.kill;
// The two eyes disappeared.
p.join;
```

urbiscript
Session

All GNU/Linux graphical application can be launched in the same way. You can also display images using the **display** command. First upload some image to Jazz filesystem using **scp**.

```
$ scp myimage.jpg gostai@A1111FRPA001001:
```

Shell
Session

then display the image from urbiscript:

urbiscript
Session

```
var p = Process.new("display",
                    ["-geometry", "800x480+-1+24", "/home/gostai/myimage.jpg"]);
[00897748] Process display
p.run;
sleep(10s);
p.kill;
p.join;
```

30.2.10.2 Command Line

You can also launch graphical commands directly from shell session, but you must first set the DISPLAY environment variable as follow:

Shell Session

```
gostai@A1111FRPA001001:~$ export DISPLAY=:0.0
gostai@A1111FRPA001001:~$ xeyes -geometry 800x500
```

30.2.11 Sonars

Jazz robot includes 8 sonars sensors. are located in front of the robot with a constant spacing in order to detect any obstacle. One is located in the torso of the robot to detect desk and the last one is located in the back of the robot. You can access those sensors using an array containing every values with this order: [left1, left2, left3, torso, right3, right2, right1, back]

30.2.11.1 urbiscript Interface

urbiscript Session

```
robot.body.sonars.val;
[00030042] [4.45448, 4.87494, 2.4531, 5.31701, 2.4292, 1.61747, 1.15333, 0.133563]
```

30.2.11.2 Example

urbiscript Session

```
/*
 * Stop the robot when sonar sensor values are less than 45 centimeters.
 */
at (watch (robot.body.sonars.val)?)
{
    var i = 0;
    for (var d in robot.body.sonars.val)
    {
        if (i <= 6 && d <= 0.45)
            robot.stop();
        i++;
    };
}
```

30.2.12 IRs

The robot includes 4 IR sensors designed specifically to detect desk and stairs that it have to avoid. Two of them are pointing to the ground so when the distance grow you should be able to know there is a hole in front of the robot. And the last two are pointing to the sky so when their value become short you can assume that there is an obstacle in front (like a chair or a desk that sometimes are too close to be detected with the sonars).

You can access an array containing the 4 sensors' values in this order: [upLeft, upRight, floorLeft, floorRight].

30.2.12.1 urbiscript Interface

```
robot.body.irs.val;
[00914315] [0.645662, 0.61491, 1.5, 1.5]
```

urbiscript
Session

30.2.12.2 Example

```
/*
 * stop the robot when IR sensors value are above (or under) 1 meter
 */
at (watch (robot.body.irs.val)?)
{
    var i = 0;
    for (var d in robot.body.irs.val)
    {
        if (i <= 1 && d >= 1
            || i >= 1 && d <= 1)
            robot.stop();
        i++;
    };
}
```

urbiscript
Session

30.2.13 Laser

Jazz has an Hokuyo laser (not available in all version of Open Jazz) accessible with “robot.body.laser”.

30.2.13.1 robot.body.laser

- **angleMax**
the angle of the last ray (in radian)
- **angleMin**
the angle of the first ray (in radian)
- **distanceMax**
the maximal distance from which obstacles can be detected
- **distanceMin**
the minimal distance from which obstacles can be detected
- **resolution**
angular step between two consecutive rays (in radian)
- **val**
an array of distances (in meter) provided by the laser from right side to left side

30.3 urbiscript Library

30.3.1 AlsaMicrophone

The AlsaMicrophone uobject is a driver UObject that can be used to communicate from urbiscript to the microphone device thanks to the ALSA (Advanced Linux Sound Architecture) library. The AlsaMicrophone prototype has only one slot available:

- **new(*device*)**
Create a new instance of the AlsaMicrophone. Takes the alsa microphone *device* name as parameter.

Using `new` you create child instances of AlsaMicrophone. These new prototypes have the following slots:

- **val**
Binary value corresponding to the sound captured by the microphone.
- **setup(*channels*, *rate*, *sampleSize*, *period*)**
Setup sound format.

30.3.2 AlsaSpeaker

The AlsaSpeaker uobject is a driver UObject that can be used to communicate from urbiscript to the speaker device thanks to the ALSA (Advanced Linux Sound Architecture) library. The AlsaSpeaker prototype has only one slot available:

- **new(*device*)**
Create a new instance of the AlsaSpeaker. Takes the alsa speaker *device* name as parameter.

Using `new` you create child instances of AlsaSpeaker. These new prototypes have the following slots:

- **bufferSize**
The maximum buffer size.
- **clear()**
Empty the sound buffer
- **closeDelay**
Close device if no sound is available for this duration
- **latency**
Latency (alsa buffer size) in samples.
- **pollFD()**
- **remain**
The amount of time remaining to play in the speaker sound buffer (expressed in *ms* as a default unit).
- **setup(*channels*, *rate*, *sampleSize*)**
Setup sound format. If not called, the format of the first received sample is used.
- **val**
The speaker value, expressed as a binary, in the format given by the binary header during the assignment.

30.3.3 EchoCanceller

This UObject remove the sound signal played by the speaker, from the sound signal captured by the microphone. The EchoCanceller prototype has only one slot available:

- **new()**
Create a new instance of the EchoCanceller.

Using `new` you create child instances of EchoCanceller. These new prototypes have the following slots:

- **aggressive_mode**
Boolean specifying if the echo canceller should function in aggressive or moderate mode.
- **calibrating**
Set this boolean to *true* to start **delay** calibration.
- **delay**
Delay estimate for sound card (expressed in *ms*).
- **energy_peak**
Energy peak used for delay calibration.
- **load**
Enable or disable echo canceller.
- **micro**
Input port to branch the microphone *val* slot in.
- **speaker**
Replace your speaker *val* slot by this slot, so that every sound that need to be played on your system goes through this echo canceller first.
- **speakout**
Branch this slot to your real speaker *val* slot (so that the sound after going through this uobject, then gets played)
- **val**
Binary value corresponding to the sound captured by the microphone, after processing by the echo canceller.

30.4 urbiscript Interfaces

An interface describes some aspects of a type of device, by specifying the slots and methods that implementations must provide. Each child node of the component hierarchy should implement at least one interface.

In short, interfaces are standard Urbi objects that components can inherit from to declare that they have some functionalities.

The following pages describe interfaces defined for Jazz and used by some of Jazz objects.

30.4.0.1 AutomaticGainControl

Interface for Automatic Gain Control (AGC) algorithm.

- **load**
Enable or disable Automatic Gain Control
- **level**
level the sound should reach on average after processing by AGC.
- **max_gain**
Maximal gain the AGC can apply.
- **increment**
Maximal gain increase in dB/second.
- **decrement**
Maximal gain decrease in dB/second.

- **gain**
Current gain applied by AGC.

30.4.0.2 Denoiser

Interface for Denoiser (remove noise from a sound signal) algorithm.

- **load**
Enable or disable denoising
- **reduction**
Set the level of denoising in dB

30.5 Troubleshooting

30.5.1 Get Jazz Version

The version of Jazz software which is installed on your robot is displayed when you connect to your robot using `netcat`, Gostai Console or Gostai Lab.

Shell Session

```
$ rlwrap nc A1111FRPA001001 54000
[00732235] *** *****
[00732235] *** Jazz version 2.10.2 rev. a2aa56e
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** Urbi SDK version 2.7.2 rev. 7a6a48f
[00732235] *** Copyright (C) 2004-2011 Gostai S.A.S.
[00732235] ***
[00732235] *** This program comes with ABSOLUTELY NO WARRANTY. It can
[00732235] *** be used under certain conditions. Type 'license;',
[00732235] *** 'authors;', or 'copyright;' for more information.
[00732235] ***
[00732235] *** Check our community site: http://www.urbiforge.org.
[00732235] *** *****
```

Here we see that Jazz version is ‘2.10.2 rev. a2aa56e’ and Urbi SDK version is ‘2.7.2 rev. 7a6a48f’.

30.5.2 Access Jazz Logs

There are several scripts available to view Jazz logs: `tail-outlog.sh` and `tail-errlog.sh` allow you to monitor in live the error and output log streams, and `less-outlog.sh` and `less-errlog.sh` allow you to display and navigate through the complete error and output logs. To quit `tail-*log.sh` you should send a terminate signal, by pressing `Ctrl+c` (or `Ctrl+break` on certain systems). To quit `less-*log.sh` simply press the `q` key.

Shell Session

```
gostai@A1111FRPA001001:~$ tail-errlog.sh
Wed 2011-05-25 14:28:45 CEST      [     Jazz.parseopt      ] profile  'teleop-rev03' loaded
Wed 2011-05-25 14:28:45 CEST      [     Jazz.real_devices.adl ] Loading...
Wed 2011-05-25 14:28:45 CEST      [     Jazz.real_devices.adl ] ADL start
Wed 2011-05-25 14:28:45 CEST      [     Jazz.real_devices.adl ] Loaded...
Wed 2011-05-25 14:28:45 CEST      [Jazz.real_devices.shared-camera] Loading...
^C
```

30.5.3 Debug your Urbi Code

30.5.3.1 Monitor urbiscript Job Execution

To get some info on the urbiscript executing in the Urbi runtime you can use `System.stats` (display the cycle execution stats) and `System.ps` (list the urbiscript jobs).

urbiscript Session

```
// Display the cycle stats (recorded on a 10s time frame)
System.resetStats;
sleep(10s);
System.stats;
[00198274] ["cyclesStdDev" => 0.00219118, "cyclesVariance" => 4801.27,
"cyclesMean" => 0.00163504, "cyclesMax" => 0.014414, "cyclesMin" => 0.000171,
"cycles" => 2110]

// Display current urbiscript jobs
System.ps;
[00215160] *** Job: shell
[00215160] *** State: idle (frozen) (side effect free)
[00215160] *** Time Shift: 0.026387ms
[00215160] *** Tags:
[00215160] *** Tag<Lobby_216>
[00215160] *** Job: event_224
[00215160] *** State: waiting
[00215160] *** Tags:
[00215160] *** Tag<Lobby_98>
[00215160] *** Tag<tag_104>
[00215160] *** Tag<tag_102>
[00215160] *** Tag<tag_152>
[00215160] *** Backtrace:
[00215160] *** rewrite/desugarer.cc:324.6-23: sleep
[00215160] *** /jazz/share/jazz/services/urbi/fsm.u:289.15-22: emitDone
[00215160] *** /jazz/share/jazz/services/urbi/fsm.u:476.16-31: enter
[00215160] *** /usr/local/stow/urbi-sdk/share/gostai/urbi/control.u:18.5-26: spawn
[00215160] *** /jazz/share/jazz/services/urbi/fsm.u:476.16-31: detach
[00215160] *** /jazz/share/jazz/services/urbi/fsm.u:436.11-52: applyTransition
[00215160] *** /jazz/share/jazz/services/urbi/fsm.u:234.13-37: transit
```

30.5.3.2 Monitor UObjects Execution Time

There is a statistic gathering tool available from urbiscript. Enable it with `uobjects.enableStats`. Reset counters by calling `uobjects.clearStats`. `uobjects.getStats` returns a `Dictionary` of all bound C++ functions called, including timer callbacks, along with the average, min, max call durations, and the number of calls.

```
// In this example, we display the execution time of UObject
// components in a 10 second time frame.
function displayStats()
{
    var sorted = getStats.asList.sort(function(obj1, obj2) {
        obj1[1][0]*obj1[1][3] < obj2[1][0]*obj2[1][3]
    });
    echo("Exec time (avg, min, max) (ms) | #calls | UObject name");
    for (var obj : sorted)
        echo("%4s, %4s, %5s | %3s | %s" % (obj[1] + [obj[0]]));
}
enableStats(true);
clearStats;
sleep(10s);

displayStats;
[00895231] *** Exec time (avg, min, max) (ms) | #calls | UObject name
[00895236] *** 106, 102, 115 | 10 | uob_0xb783b5d8.load --> uob_0xb783b5d8
[00895238] *** 41, 24, 358 | 150 | VideoIn_0xb70ba2b8.valJPEG --> uob_0xb7141328
[00895240] *** 230, 197, 423 | 40 | ifconfig_0xb70972e8 update
[00895243] *** 26, 25, 236 | 380 | uob_0xb78548d8.sharp2 --> uob_0xb7199408
[00895245] *** 26, 25, 95 | 380 | uob_0xb78548d8.sharp1 --> uob_0xb7199408
[00895247] *** 248, 229, 318 | 40 | Iwconfig_0xb7827278 update
[00895249] *** 27, 25, 65 | 380 | uob_0xb78548d8.sharp3 --> uob_0xb7199408
[00895252] *** 34, 29, 276 | 380 | uob_0xb78548d8.sharp0 --> uob_0xb7199408
```

urbiscript
Session

```
[00895254] *** 104,    16, 11699 | 380 | uob_0xb70f2628.speedAlpha --> uob_0xb70f2628
[00895264] *** 455,   412,   736 | 390 | uob_0xb7199408.inputSpeedAlpha --> uob_0xb7199408
[00895266] *** 503,   456,   745 | 390 | uob_0xb7199408.inputSpeedDelta --> uob_0xb7199408
[00895268] *** 1067,    30, 10795 | 380 | uob_0xb70f2628.speedDelta --> uob_0xb70f2628
[00895271] *** 1406,   249, 12163 | 380 | uob_0xb78548d8.sonars --> uob_0xb7199408
[00895272] *** 2812, 1557, 13606 | 380 | uob_0xb78548d8
```

With this information you can check whether your UObjects take too long to execute.

30.5.4 Enable Core Dump

The Linux kernel is able to generate core dumps when a program crashes. These core-dump files can then be given to `gdb` to debug the program. To enable core dumps in the Linux kernel, `ssh` on the robot and do:

Shell Session

```
gostai@A1111FRPA001001:~$ sudo ulimit -c unlimited
gostai@A1111FRPA001001:~$ ulimit -c unlimited
gostai@A1111FRPA001001:~$ echo 1 > /proc/sys/kernel/core_uses_pid
```

30.5.5 Monitor Jazz Processor and Memory Load

Log in with `ssh` and use `top` or `htop` to view Jazz load. If your robot is overloaded the Urbi runtime cannot guarantee a normal behavior.

30.5.6 Monitor Jazz Hard-Drive Space

Log in with `ssh` and use '`df -h`'. It displays the amount of memory used and available (look at the '`/dev/sda1`' line).

30.5.7 I/O Issues when Powering Robot Off

If you edit files directly on the robot (using `ssh`), the filesystem cannot guarantee that your changes will be effectively written to memory if you shutdown the robot with the on/off power button just after having made your changes. If you need to stop or restart your robot, either:

- use the '`sync`' linux command before you shutdown your robot with the on/off power button.
- use soft shutdown ('`sudo halt`') or reboot commands ('`sudo reboot`').

30.6 ROS support

30.6.1 About ROS

ROS (Robot Operating System) is a set of tools and libraries designed to provide a common framework for robotic application developers. There are already thousands of compatible applications available in various domains such as computer vision, navigation, visualization tools...

30.6.2 ROS support in Jazz

Jazz comes with complete ROS compatibility. The currently used ROS version is 'electric'. A subset of the ROS distribution is installed in the `/opt/ros` directory of the Jazz hard drive.

30.6.3 Enabling and configuring ROS service

ROS support in Jazz is implemented as a Jazz service and is disabled by default. Refer to section [Section 30.1.10](#) for instructions on how to enable and disable services.

By default, a ROS master is started on the robot when the service is enabled. You can specify a different ROS master by setting the `config.ros.master_uri` configuration variable:

```
config.ros.master_uri = "http://my_computer:11311";
// Call _save to make the change persistent.
config.ros._save;
```

urbiscript
Session

30.6.4 Predefined topics

The Jazz ROS service maps each device of the robot to a ROS Topic with similar name. The following table lists the most important topics:

Urbi name	ROS topic	Dir	ROS message type
<code>jazz.body.irs</code>	/robot/body/irs	Pub	sensor_msgs/LaserScan
<code>jazz.body.sonars</code>	/robot/body/sonars	Pub	sensor_msgs/LaserScan
<code>jazz.body.laser</code>	/robot/body/laser	Pub	sensor_msgs/LaserScan
<code>jazz.camera</code>	/robot/body/head/camera	Pub	sensor_msgs/Image
<code>jazz.body.head.pitch</code>	/robot/body/head/pitch_out	Pub	std_msgs/Float32
	/robot/body/head/pitch_in	Sub	std_msgs/Float32
<code>jazz.body.head.yaw</code>	/robot/body/head/yaw_out	Pub	std_msgs/Float32
	/robot/body/head/yaw_in	Sub	std_msgs/Float32
	/robot_in	Sub	geometry_msgs/Twist
<code>jazz</code>	/robot_out	Pub	nav_msgs/Odometry
	/tf	Pub	tf/tfMessage

The sonar array and the laser if present are accessible as standard LaserScan data. Robot odometry is available as either nav_msgs/Odometry messages in the /robot_out topic or standard tf transforms on the /tf topic.

All sensor messages with a header are given in the `base_footprint` frame. tf messages from the odometry are published on the standard /tf topic, performing the `/base_footprint -> /odom` transform.

Head yaw and pitch are mapped to two different topics, since they have a sensor part and an actuator part.

30.6.5 Using the navigation stack with ROS

If your Jazz is equipped with a laser scanner, Jazz is compatible with the ROS [navigation stack](#).

Navigation components can run on the Jazz CPU, or as remote components.

It is recommended to run them remotely, especially if you plan to use the web control interface at the same time to avoid overloading the Jazz CPU.

30.6.5.1 Creating a map of your environment

The `gmapping` module can be used to create a map of your environment.

Make sure you are using the same ros-master as your Jazz. If you type `rostopic list` you should see all the topics advertised by the robot.

Then run:

```
rosrun gmapping slam_gmapping scan:=/robot/body/laser
```

Shell
Session

This will start gmapping. gmapping then processes odometry and laser scan data to create a map of the environment as the robot moves.

Using the joystick (preferred) or the web interface, navigate the robot around your environment. You should avoid fast rotations as it puts too much stress on the SLAM process.

When you are done, extract the map using:

Shell Session

```
rosrun map_server map_saver
```

This will create the file `map.pgm` containing your map data.

You should then open the map in an image editor and crop it to remove the empty space. This will reduce CPU requirements for the localization task.

30.6.5.2 Autonomous navigation

You will find on the robot a ROS package named `jazz_navigation` under `JAZZ_ROOT/share/vigilant/ros` that contains a reasonable default configuration for the navigation.

You can copy the whole directory to an external computer (recommended) or run it on the robot:

Shell Session

```
# Copy the parameter files (replace myJazzName with the host name of your Jazz)
scp -r myJazzName:/usr/local/stow/vigilant/share/vigilant/ros jazz-ros
# Make the package visible to ros
export ROS_PACKAGE_PATH=$PWD/jazz-ros:$ROS_PACKAGE_PATH
# Run the navigation package
roslaunch jazz_navigation move_base.launch
```

See the ROS [navigation tutorial](#) for information about the various configuration files.

You can visualize sensor data and set navigation goals using the ROS [rviz](#) tool:

Shell Session

```
rosrun rviz rviz
```

Then load the `jazz_navigation/nav.vcg` layout file in `rviz`, which contains the proper `rviz` configuration for Jazz.

30.6.5.3 Troubleshooting

Jazz does not move when setting a navigation Goal If Jazz is docked on its charging station, move commands are disabled. You must remove Jazz from its charging station manually.

Chapter 31

Pioneer 3-DX

This UObject has been created to drive the Pioneer 3-DX robot. It uses the Aria/Arnl manufacturer library.

31.1 Getting started

31.1.1 Prerequisites

- Have a Pioneer P3-DX robot with linux or any linux distribution with MobileSim installed (<http://robots.mobilerobots.com/wiki/MobileSim>).
- Have Urbi Sdk and ARNL libraries installed (http://robots.mobilerobots.com/wiki/ARNL,_SONARNL_and_MOGS).

31.1.2 Installation

- Untar your archive file
- Move in your archive and create a config.mk file where variables UPATH and ARNLPATH are defined. For example

```
UPATH=/usr/local/gostai # path for your urbi-sdk directory  
ARNLPATH=/usr/local/Arnl # path for your arnl directory
```

Shell Session

- Run `make` to compile

31.1.3 Running

- If you are working with the simulator MobileSim, make sure it is running.
- Run ‘`make run`’.
- Connect to the Urbi server, for instance with ‘`telnet IPYouNeed 54000`’.

31.2 How to use Pioneer 3-DX robot

The following objects are defined to support the Pioneer 3-DX Robot. Please note that when loaded, ‘`p3dx.u`’ also defines the `robot` variable to enforce the conformance with the Gostai Standard Robotics API ([Chapter 26](#)).

Below, we denote read-only slots with ‘`r`’, and read-write slots with ‘`rw`’.

31.2.1 P3dx

- **body**
The `P3dx.body` class.
- **go(*d*)**
Cover *d* meters.
- **model**
Model of the robot (r).
- **moveTo(*list*)**
Try to plan a path to position *list* == [x,y,th] in absolute frame. `mapFileName` must have a valid value and `motorsLoad` must be 1.
- **name**
Name of the robot (r).
- **robotType**
Type of the robot (r).
- **serial**
Serial number of the robot (r).
- **stop**
Stop any movement.
- **turn(*r*)**
Turn *r* radians.

31.2.2 P3dx.body

- **load**
Enable the motors (rw)
- **safetyDistanceMax**
Speed is not restrained if all obstacles
- **safetyDistanceMin**
Moving is not allowed if any obstacle is closer (rw)
- **sonar**
The object `P3dx.body.sonar`.
- **wheel[left].speed**
Left wheel translation speed (r)
- **wheel[right].speed**
Right wheel translation speed (r)

31.2.3 P3dx.body.odometry

- **coveredAngle**
Total yaw angle covered (r)
- **coveredDistance**
Total distance covered (r)
- **position**
List for [x, y, z, yaw] in absolute frame (rw)

- **x**
Odometric x position (r)
- **y**
Odometric y position (r)
- **yaw**
Odometric yaw angle (r)
- **z**
Always 0 (r)

31.2.4 P3dx.body.sonar

- **load**
Enable the sonars (rw).
- **[front] [left] [left].val**
value for the front/left/left sonar (r)
- **[front] [left] [front].val**
value for the front/left/front sonar (r)
- **[front] [front] [left].val**
value for the front/front/left sonar (r)
- **[front] [front] [right].val**
value for the front/front/right sonar (r)
- **[front] [right] [front].val**
value for the front/right/front sonar (r)
- **[front] [right] [right].val**
value for the front/right/right sonar (r)
- **[right] [front].val**
value for the right/front sonar (r)
- **[right] [back].val**
value for the right/back sonar (r)
- **[back] [right] [right].val**
value for the back/right/right sonar (r)
- **[back] [right] [back].val**
value for the back/right/back sonar (r)
- **[back] [front] [right].val**
value for the back/front/right sonar (r)
- **[back] [front] [left].val**
value for the back/front/left sonar (r)
- **[back] [left] [back].val**
value for the back/left/back sonar (r)
- **[back] [left] [left].val**
value for the back/left/left sonar (r)

- **[left] [back].val**
value for the left/back sonar (r)
- **[left] [front].val**
value for the right/front sonar (r)

31.2.5 P3dx.body.laser

- **angleMax**
Angle of the first value in the robot frame (r)
- **angleMin**
Angle of the last value in the robot frame (r)
- **laserDistanceMax**
Maximum distance the laser can return (r)
- **laserDistanceMin**
Minimum distance the laser can return (r)
- **lastCaptureTimestamp**
Last time laser read its values (r)
- **load**
Connect to sick laser (rw)
- **resolution**
Angle between to laser values (r)
- **val**
181 distances given by the laser (r)

31.2.6 P3dx.body.camera

- **load**
Connect the camera mechanical aspect (rw)
- **pitch**
Pitch value for camera (rw)
- **yaw**
Yaw value for camera (rw)
- **zoom**
Zoom value for camera (rw)

31.2.7 P3dx.body.x

- **speed**
Translation speed (rw)

31.2.8 P3dx.body.yaw

- **speed**
Rotation speed (rw) are further (rw)

31.2.9 P3dx.planner

- **locFailureEvent**

Event emit-ed if localization fails. In this case `mapFileName` is set to “”.

- **mapFileName**

Path to a map file. Setting a string value will also try to start a localization task. `laserLoad` must be 1. This may fail if:

- the file is unknown;
- the robot cannot localize itself on the map (the initial position of the robot SHOULD be equal to the Home position inside the map file).

Value “” deletes the previous map and stops the localization task.

At any time, it might be set to “” if the localization task fails.

- **pathPlanningEvent**

Event emitted when path planning ends (r) with first parameter equal to:

- 0** if the path planning fails;
- 1** if it succeeds;
- 2** if it is canceled for any reason (new path, manual motion commands, change of map).

31.2.10 P3dx.body.battery

- **voltage**

Current voltage (r)

31.3 Mobility modes

Three modes are available in order to control the Pioneer 3-DX mobility:

- Use `go` and `turn` to choose a distance and an angle.
- Use `speedDelta` and `speedAlpha` to choose translation and rotation speeds.
- Use `moveTo` to select a goal on a map that must be reached thank to path planning. Note that `distMin` and `distMax` have no impact in this mode.

31.4 About units

Every physical quantity is in SI units (m, s, rad, ...).

Chapter 32

Segway RMP

The Segway *RMP* (Robotic Mobility Platform) is a robotic platform based on the Segway Personal Transporter. Urbi for Segway RMP can be downloaded from <http://www.urbiforge.org/index.php/Robots/Segway>.

32.1 RMP

The UObject RMP provides the interface from Urbi.

32.1.1 Instantiation

```
var rmp = RMP.new(period);
```

urbiscript
Session

period is the period, in seconds, at which orders will be sent and coders will be read.

32.1.2 Slots

Raw actuator variables

- **forwardSpeed**
forward speed in meters per second. Can be read to (behaves as a sensor), or written to (actuator).
- **yawSpeed**
yaw speed in radian per second. Can be read and written to.

Raw sensors Current speed and coder values.

- **forward**

- **left**

- **leftSpeed**

- **right**

- **rightSpeed**

- **yaw**

Calibration Urbi unit per coder unit coefficients.

- forwardCoeff
- forwardSpeedCoeff
- leftCoeff
- leftSpeedCoeff
- rightCoeff
- rightSpeedCoeff
- yawCoeff
- yawSpeedCoeff

The file ‘usegway.u’ contains approximate values for meter, seconds, radian units.

32.2 GSRAPI compliance

The file ‘rmp.u’ will initialize the RMP UObject and give it a Gostai Standard Robotic API compliant interface.

Chapter 33

Spykee

The *Spykee* is a WiFi-enabled robot built by Meccano (known as Erector in the United States). It is equipped with a camera, speaker, microphone, and moves using two tracks.

33.1 Installing Urbi on the Spykee

To enable Urbi in the Spykee, you must update it with a new firmware. The Urbi firmware can be downloaded from <http://www.gostai.com>.

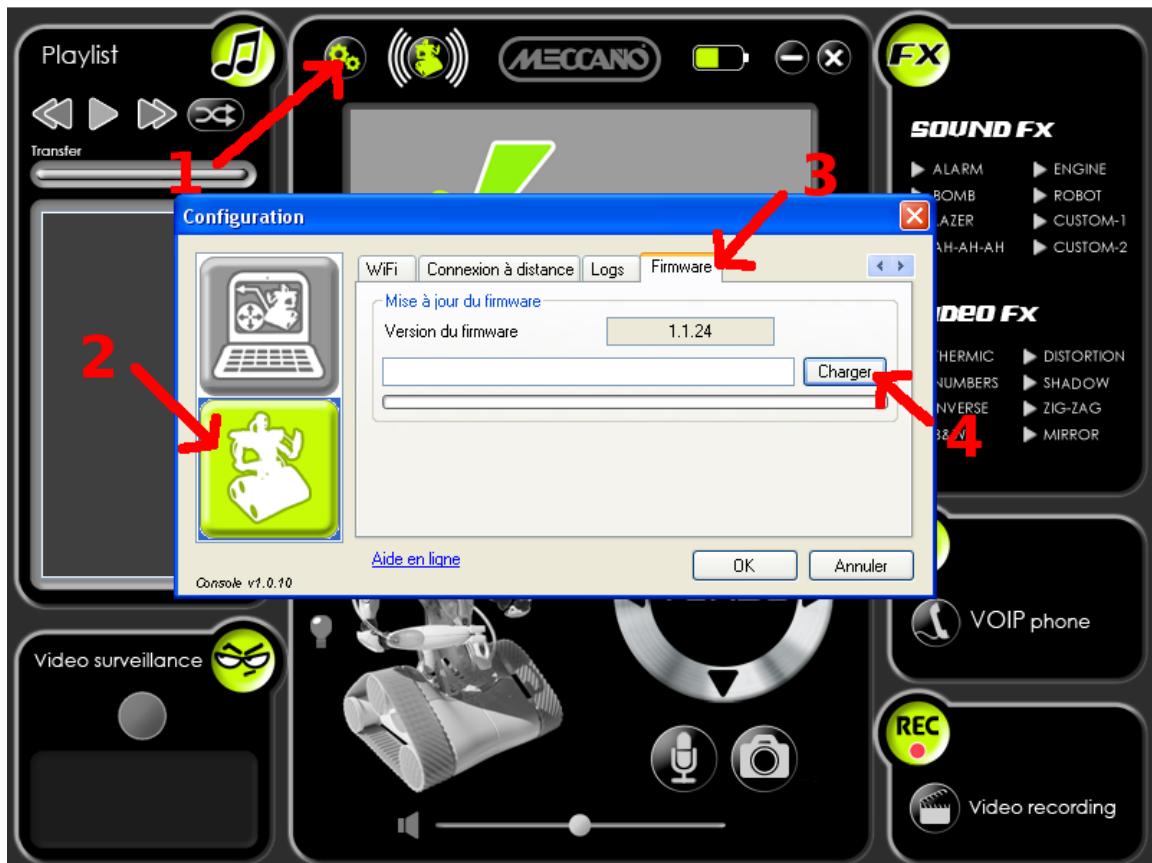


Figure 33.1: Installing the Spykee interface

Once obtained, use the Spykee interface. To install it (see [Figure 33.1](#)): Download it from <http://spykeeworld.com>, start the interface, connect to your robot, click on the “settings” button, then on the Spykee image on the left. Then go to the “Firmware” tab, and click the “Load” button. Select the firmware file you downloaded earlier, and press OK. Leave the Spykee

on its base until it reboots, the update process will take a few minutes. Do not reboot the Spykee yourself while update is in progress or you might render it useless.

With the Urbi firmware, the Spykee should continue to work as usual: you can still control it using the Spykee interface. But you can now also control it with Urbi.

33.2 Device list

The following standard devices are available on the Spykee:

- **trackL, trackR (Motor)**
The two tracks. The `val` slot can be read or set to change each track speed, from -100 to 100.
- **ledL, ledR, headLed (Led)**
left and right led pairs, and camera light. On if the `val` slot is set to 1.
- **camera (VideoIn)**
Spykee camera. The `val` slot contains the last received image in jpeg format.
- **speaker (AudioOut)**
Sound speaker.
- **micro (AudioIn)**
Microphone.
- **robot (Mobile)**
Provides the standard functions `go()` and `turn()`.

The following non-standard functions are also present:

- **speaker.playSound(*id*)**
Play Spykee sound sample `id`, from 0 to 15.
- **mp3.play(*stream*)**
Play mp3 file or http stream `stream`.
- **mp3.stop()**
Stop playing current stream or file.
- **Spykee.setConfigValue(*key*, *value*)**
Set persistent configuration entry `key` to `value`. Beware that the configuration sector only have a few kilobytes of space, do not put too many or to big values here.
- **Spykee.getConfigValue(*key*)**
Retrieve persistent value associated with `key`. Do not call this function two often as it is very costly.
- **Spykee.findBase()**
Ask the Spykee to look for its base and station on it.
- **Spykee.cancelFindBase()**
Abort the previous order.
- **Spykee.chargeStop()**
Ask the Spykee to stop charging and exit its charging station.
- **Spykee.power**
Remaining energy in the battery, from 0 to 100. The value may be inaccurate or outside the range while the Spykee is charging.

33.3 Dynamically loaded modules

At start-up, the Spykee tries to connect to the `spykee.gostai.com` website to fetch a list of additional features available for download. It sets up the system in a way that the first attempt to use one of those additional features will download and load the required objects.

The location where the modules are fetched can be changed by setting the `loader_host` and `loader_dir` configuration values (using `Spykee.setConfigValue()`). Default values are `spykee.gostai.com` and `/modules/`.

The following modules are currently available:

33.3.1 Clapper

The Clapper module detects and counts short and loud sounds, such as the one made by clapping hands. It can be instantiate by typing:

```
var Global.clap = Clapper.new("micro.val", 0.2, 200, 800, 4000);
```

urbiscript
Session

The second parameter is the volume threshold above which a clap is detected. The third and fourth parameters are the minimum and maximum delay between claps, in milliseconds. The last argument is the maximum duration of a sound sequence.

Each time a clap sequence ends, the `val` slot of the object is set to the number of claps heard, then reset to 0. Additionally, if `emitEvent` is true, the event `sequence` is emitted with one argument: the number of claps heard.

```
clap.emitEvent = 1;
// Play sound sample number 3 when two successive claps are heard.
tag:at(clap.sequence?(2)) speaker.playSound(3);
// Stop the previous behavior.
tag.stop;
```

urbiscript
Session

Chapter 34

Webots

This chapter documents the use of Urbi in the Webots simulation environment.

34.1 Setup

This section describes how to setup the software on your computer and give you the basis to use Urbi for Webots.

34.1.1 Installation

The setup process depends on your OS, please follow the corresponding instructions.

34.1.1.1 Linux

To install Urbi for Webots in your Webots on Linux, you must download an archive of the form ‘`urbi-for-webots-version-linux-x86-gcc4-release.tar.bz2`’. You may choose an other archive format than `.tar.bz2`, but you will need to slightly adjust the following instruction accordingly.

The archive is structured to blend into the Webots hierarchy, as follow:

- ‘`urbi-for-webots-version/projects/default/controllers/urbi2.0/`’
Urbi SDK and Webots controller.
- ‘`urbi-for-webots-version/projects/packages/urbi2.0/share/robot-name/`’
Shared urbiscript robot behaviors.
- ‘`urbi-for-webots-version/projects/packages/urbi2.0/controllers/robot-name/`’
Demonstration controllers for each robot.
- ‘`urbi-for-webots-version/projects/packages/urbi2.0/worlds/`’
Demonstration worlds.

All you need to do is extract it into your Webots directory, minus the `urbi-for-webots-version` containing directory, so as to merge the packaged `project` directory with your Webots’ `project` directory.

This can be easily done with the following `tar` command:

```
tar -xvf urbi-for-webots.tar.bz2 -C /path/to/webots --strip-components 1
```

Shell
Session

Otherwise, you can also merge the directory after extracting the archive:

```
cp --archive urbi-for-webots/project /path/to/webots
```

Shell
Session

You can check your installation works correctly by starting a test world:

Shell
Session

```
cd /path/to/webots
./webots projects/packages/urbi2.0/worlds/aibo-ball-tracking.wbt
```

You should see an Aibo looking at a moving ball.

34.1.1.2 Mac OS X

Urbi for Webots is not available under Mac OS X anymore for now. We hope to fix this very shortly.

34.1.1.3 Windows

Urbi for Webots is not available under windows anymore for now. We hope to fix this very shortly.

34.1.2 License

34.1.2.1 Evaluation mode

If you don't provide a valid license file to the Urbi controller, it will be launched in *evaluation mode*. The only difference with the normal version is that the controller will exit after five minutes, thus letting you test all the features of the registered version during this amount of time. In evaluation mode, the Urbi controller will display the following message in the Webots console:

```
*****
* URBI: Invalid or missing license.
* Urbi for Webots will quit after five minutes.
*****
```

In order to run Urbi for Webots without constraints you must provide it with a license file, as described in the following section.

34.1.2.2 Setting up a license

Providing a valid Urbi for Webots license will enable you to run Urbi controllers for an unlimited amount of time. The license is a file named ‘`urbi.key`’. You simply need copy it in the ‘`webots/resources/`’ directory of your Webots installation.

The Urbi for Webots licensing system is based on the Webots licensing system: your Urbi for Webots key is attached to your Webots key. The procedure will depend on whether your Webots license is based on a ‘`webots.key`’ file or an USB dongle.

If you have any problem setting up your Urbi for Webots license, feel free to contact us at contact@gostai.com.

Urbi licensing with a ‘`webots.key`’ Webots license: If you use a file-based Webots license, please send us your ‘`webots.key`’ on contact@gostai.com so as we can generate your ‘`urbi.key`’. In this case, you need to put your ‘`webots.key`’ together with your ‘`urbi.key`’ in the ‘`webots/resources/`’ directory.

Urbi licensing with an USB dongle Webots license: If you use an USB dongle based license, please send us your dongle number available in the help/about Webots menu to contact@gostai.com so as we can generate your ‘`urbi.key`’.

34.2 Using Urbi as a controller in your Webots worlds

This section will show you how to control your own simulation with Urbi controllers.

A robot controlled by Urbi in Webots uses a controller which is actually a standard Urbi server, with custom UObjects plugged in. This mean you can connect to the Urbi server and interact with the robot just like you would with the actual Urbi server of the robot. Note however that since the Urbi server is started by webots, we have no way to interact with it on the standard input, and need to connect to it via the network. Please refer to [Chapter 3](#) to learn how to interact with an Urbi server via the network.

You can try it out with one of the bundled demonstration worlds. For instance, run in Webots the ‘aibo-ball-tracking.wbt’ world located in the ‘`projects/packages/urbi2.0/worlds/`’. You can then connect on the 54000 port and control the Aibolike a regular one.

34.2.1 Using the default urbi-2.0 controller

To use an urbi controller in your Webots world, first define an `urbi-2.0` controller for your webots node:

```
DEF MYNODE Robot
{
    ...
    controller "urbi-2.0"
}
```

This will start an Urbi server to interact with your robot. This server is however useless since it doesn’t run any specific behavior and you cannot interact with it. You can ask the Urbi server to listen for incoming connection, so as you can connect to it and control the robot interactively. You can also load urbiscript files in the server to run them at start-up. You can of course do both.

To do this, simply pass arguments to the controller with the `controllerArgs` Webots attribute. Available arguments are the exact same as the `urbi` program, as described in [Section 22.3](#).

Listening for incoming connections: pass a `--port` option to the controller.

```
DEF MYNODE Robot
{
    ...
    controller "urbi-2.0"
    controllerArgs "--port 54000"
}
```

Loading an urbiscript file: pass a `--file` option to the controller. In addition to standard directories, scripts are also searched in the ‘`projects/packages/urbi-2.0/share/`’ directory of your webots installation. For instance, you could create a ‘`my_behavior/my_script.u`’ script in this directory, and use the following `controllerArgs`.

```
DEF MYNODE Robot
{
    ...
    controller "urbi-2.0"
    controllerArgs "--file my_behavior/my_script.u"
}
```

34.2.2 Defining custom Urbi controllers

A different approach would be to define your own Urbi-based controller. If you create a controller in Webots' controllers directory and create a 'global.u' file in it, Webots will automatically recognize it as an Urbi controller and start the 'urbi-2.0' controller. The 'global.u' and potential 'local.u' will be automatically loaded by the Urbi kernel, as documented in [Section 22.2](#).

This approach is equivalent to using the default 'urbi-2.0' controller and passing it file to load, but enables you to define custom, reusable controllers for your robots. For instance, you could create a 'my-project/controllers/my-robot/' controller, with a 'global.u' file in it that load all your robots objects (motors, sensors, ...). To use an Urbi controller for your robot, you would then simply have to use controller "my-robot" in your world file. If you used the default 'urbi-2.0' controller, you would have to add

```
controllerArgs "--file some-dir/my-robot-bindings.u"
```

to get a functional controller for your robot.

34.3 Binding your own robots with Urbi for Webots

Binding your own robot model with Urbi for Webots is extremely similar to binding an actual robot with Urbi, except the UObjects (device drivers) are already written for Webots generic devices. It thus mainly consists in writing an initialization urbiscript file that instantiate those Urbi objects for your robots devices (motors, sensors, ...). For instance, a binding for a minimalist robot that only has a servomotor and a distance sensor could be:

urbiscript
Session

```
var motor = Servo.new("webots_motor_node_name");
var distance = DistanceSensor.new("webots_sensor_node_name");
```

Note that, whenever possible, your Webots binding should be strictly identical to the actual robot binding (if any) so as the same Urbi code will work in the simulator and in reality. Additionally, you should whenever possible respect the Gostai Standard Robotics API (see [Chapter 26](#)) to be compatible with standard Urbi components.

34.4 Built-in robots and worlds

Urbi for Webots comes with several built-in demonstration robots and worlds. Every included robot has a binding file named 'projects/packages/urbi-2.0/share/robot-name/robot-name.u' and a custom controller located in 'projects/packages/urbi-2.0/controllers' whose 'global.u' start-up file loads the binding. Thus, using controller "robot-name" in the webots world file automatically starts an Urbi kernel with the right binding to control the robot.

Here is a description of the built-in worlds:

- 'aibo-ball-tracking.wbt'

This world demonstrates the Urbi classical Aiboball tracking behavior. It uses two Urbi controllers:

- The first one controls the Aiboball runs the tracking behavior. It uses the 'aibo' controller and the 'aibo/ball-tracking.u' script. You can connect to this controller on port 54000.
- The second controls the ball. It uses the generic 'urbi-2.0' controller and moves the ball around automatically with the 'aibo/move-ball.u' script thanks to the supervisor API (see [Section 34.5.2](#)).

- 'aibo-soccer.wbt'

This empty world simply provide and Aiboon a soccer field with a ball. It uses the 'aibo' controller. You can connect to it on the 54000 port to control the Aiboor program behaviors.

- ‘e-puck-wander.wbt’

This world presents an e-puck robot wandering around randomly. It uses the ‘e-puck’ controller and the ‘e-puck/wander.u’ script for the wandering behavior. You can connect to it on port 54000.

- ‘pioneer-wander.wbt’

This world presents a Pioneer robot wandering around randomly. It uses the ‘pioneer’ controller and the ‘pioneer/wander.u’ script for the wandering behavior. You can connect to it on port 54000.

34.5 Webots UObjects

34.5.1 Robot devices UObjects

These uobjects are the Urbi API for Webots robots devices such as motors or sensors. They all implement at least an interface from the Gostai Standard Robotics API (see [Chapter 26](#)). Please refer to these interfaces to discover most functionalities of the object. Only additional slots are documented here.

All objects are constructed with a `node_name` argument, which is the name of the corresponding node in the Webots world.

34.5.1.1 Accelerometer

While Webots accelerators all have three components, Urbi prefer to differentiate every component in one linear accelerometer.

Gostai Standard Robotic API interface: [Interface.AccelerationSensor](#)

Constructor: `Accelerometer.new(node_name, component_name)`

- `node_name` Name of the Webots node.
- `component_name` The component of the Webots accelerometer to use. Must be "x", "y" or "z".

Note: Since there is only one device for the three components in Webots, disabling one of the component (by setting the `load` slot to `false`) will disable all three components.

34.5.1.2 Camera

Gostai Standard Robotic API interface: [Interface.VideoIn](#)

Constructor: `Camera.new(node_name)`

- `node_name` Name of the Webots node.

Additional slots

- `near`

Permissions read only.

Type Float.

Description Distance of the camera to the near OpenGL clipping plane.

- `far`

Permissions read only.

Type Float.

Description Distance of the camera to the far OpenGL clipping plane.

Note: The Webots camera won't appear in the webots interface until the corresponding UObject has been instantiated. You can create only one camera Urbi Object by Webots camera device.

34.5.1.3 Differential Wheels

Differential wheels in Webots represent a pair of wheels. They are represented as separated motors in Urbi.

Gostai Standard Robotic API interface: [Interface.RotationalMotor](#)

Constructor: `DifferentialWheels.new(node_name, left)`

- *node_name* Name of the Webots node.
- *left* A Boolean: attach to the left wheel if true, to the right otherwise.
- `encoder`

Permissions read / write.

Type Float.

Description Encoders are counters incremented when a wheel turns, according to the *encoderResolution* Webots attribute of the DifferentialWheels. Setting the encoder value will not rotate the wheel, it will simply reset it to the given value.

Note: The differential wheels won't work correctly unless you instantiate both left and right objects.

34.5.1.4 Distance Sensor

Gostai Standard Robotic API interface: [Interface.DistanceSensor](#)

Constructor: `DistanceSensor.new(node_name)`

- *node_name* Name of the Webots node.

Additional slots

- `factor`

Permissions read / write.

Type Float.

Description Multiplier factor used to compute the distance.

34.5.1.5 Led

Gostai Standard Robotic API interface: [Interface.Led](#)

Constructor: `Led.new(node_name)`

- `node_name` Name of the Webots node.

34.5.1.6 Servo

Gostai Standard Robotic API interface: `Interface.RotationalMotor`

Constructor: `Servo.new(node_name, force_feedback)`

- `node_name` Name of the Webots node.
- `force_feedback` Whether to enable the reading of the force applied to the joint.

Additional slots

- `force`

Permissions read only.

Type Float.

Description Force currently deployed by the motor to achieve the desired motion. Available only if `force_feedback` was turned on at construction time.

Fixme: This is named `torque` in the standard, and should be renamed accordingly.

34.5.1.7 Touch Sensor

Gostai Standard Robotic API interface: `Interface.TouchSensor`

Constructor: `TouchSensor.new(node_name)`

- `node_name` Name of the Webots node.

34.5.2 Supervisor API UObjects

The supervisor UObjects are only available from a supervisor robot (i.e. from the controller of a supervisor node in Webots). They enable to control the simulation in an omnipotent manner.

34.5.2.1 Label

Constructor: `Label.new(text)`

- `text` The displayed text.

Slots

- `txt`

Permissions read / write.

Type String.

Description Displayed text.

- `x, y`

Permissions read / write.

Type Float.

Description Coordinates of the label on the screen.

- `r, g, b, a`

Permissions read / write.

Type Float.

Range 0, 1.

Description Red, blue, green and alpha components of the label's text color.

- `size`

Permissions read / write.

Type Float.

Description Size of the displayed text.

34.5.2.2 Manipulate Node

Constructor: `ManipulateNode.new(node_name)`

- `node_name` Name of the Webots node to manipulate. Webots' ManipulateNode objects only apply to solid objects.

Slots

- `tx, ty, tz`

Permissions read / write.

Type Float.

Description Applied translation on the three axis.

- `ax, ay, az`

Permissions read / write.

Type Float.

Description Applied rotation on the three axis.

34.5.2.3 Simulation Controller

Constructor: `SimulationController.new()`

Slots

- `physicsReset()`

Send a request to the simulator process, asking to stop the movement of all physics enabled solids in the world. It means that for any Solid node containing a Physics node, the linear and angular velocities of the corresponding body is reset to 0, hence the inertia is stopped. This function is especially useful when resetting a robot at an initial position from which no initial inertia is required. This function resets the seed of the random number generator used in Webots, so that noise based simulations can be reproduced identically after calling this function.

- `revert()`

Send a request to the simulator process, asking to reload the current world immediately. As a result of reloading the current world, the supervisor process and all the robot processes are terminated and restarted. You might want to save some data in a file from your supervisor program to be able to reload it when the supervisor controller restarts.

- **quit()**

Send a request to the simulator process, asking to terminate and quit immediately. As a result of terminating the simulator process, all the controller processes, including the calling supervisor controller process will terminate.

Part VI

Tables and Indexes

About This Part

This part contains material about the document itself.

Chapter 35 — Notations

Conventions used in the type-setting of this document.

Chapter 36 — Release Notes

Release notes of Urbi SDK.

Chapter 37 — Licenses

Licenses of components used in Urbi SDK.

Chapter 38 — Bibliography

References to other documents such as documentation, scientific papers, etc.

Chapter 39 — Glossary

Definition of the terms used in this document.

Chapter 40 — List of Tables

Index of all the tables: list of keywords, operators, etc.

Chapter 41 — List of Figures

Index of all the figures: snapshots, schema, etc.

Chapter 42 — List of Figures

An index of concepts, objects, and some selected routines.

Chapter 35

Notations

This chapter defines the *notations* used in this document.

35.1 Words

- **code**
A *piece of code* (urbiscript, Java, C++ ...).
- **comment**
A *comment* in some programming language. For instance, `/* hello /* world */ ! */` is a comment in urbiscript.
- **environment-variable**
An *environment variable* name, e.g., PATH.
- **'file-name'**
A *file name*.
- **keyword**
A *keyword* in some programming language. For instance, `watch` is an urbiscript keyword.
- **meta-variable**
Depending on the context, a *variable* name (i.e., an identifier in C++ or urbiscript), or a *meta-variable* name. A meta-variable denotes a place where some syntactic construct may be entered. For instance, in `while (expression) statement`, *expression* and *statement* do not denote two variable names, but two placeholders which can be filled with an arbitrary expression, and an arbitrary statement. For instance:
`while (!tasks.empty) { tasks.removeFront.process }.`
- **string**
A *string* in some programming language. For instance, `"Hello, world!"` is a string in urbiscript.

35.2 Frames

35.2.1 C++ Code

C++ source code is presented in frames as follows.

```
class Int
{
public:
    Foo(int v = 0)
        : val_(v)
```

C++

```

    {}

void operator(int v)
{
    std::swap(v, val_);
    return v;
}

int operator() const
{
    return val_;
}

private:
    int val_;
};

```

35.2.2 Grammar Excerpts

The *grammar fragments* are written in *EBNF (Extended Backus-Naur Form)*. The symbol `::=` separates the left-hand symbol from the right-hand side part of the rule. Infix `|` denotes alternation, postfix-`*` 0-or-more repetition, postfix-`+` 1-or-more repetition, and postfix-`?` denotes optional parts. Terminal symbols are written in double-quotes, and non-terminals in angle-brackets. Parentheses group.

The following frame defines the grammar syntax expressed in the same grammar syntax.

Grammar Excerpt

```

<grammar> ::= <rule>+
<rule> ::= <symbol> " ::= " <rhs>

<rhs> ::= <rhs>*
        | <rhs> " | " <rhs>
        | <rhs> ("?" | "*" | "+")
        | "(" <rhs> ")"
        | <symbol>

<symbol> ::= "<" <identifier> ">"
            | " " <escaped-character>* " "
            | " " <escaped-character>* " "

```

35.2.3 Java Code

Java source code is presented in frames as follows.

Java

```

import liburbi.main.*;
public class Main
{
    /// Load urbijava library.
    static
    {
        System.loadLibrary("urbijava");
    }

    public static void main(String argv[])
    {
        // Does nothing for now.
    }
}

```

35.2.4 Shell Sessions

Interactive sessions with a (Unix) shell are represented as follows.

```
$ echo toto
toto
```

Shell
Session

The user entered ‘echo toto’, and the system answered ‘toto’. ‘\$’ is the *prompt*: it starts the lines where the system invites the user to enter her commands.

35.2.5 urbiscript Sessions

Interactive sessions with Urbi are represented as follows.

```
echo("toto");
[00000001] *** toto
```

urbiscript
Session

Contrary to shell interaction (see [Section 35.2.4](#)), there is no prompt that marks the user-entered lines (here `echo("toto");`, but, on the contrary, answers from the Urbi server start with a label that includes a timestamp (here ‘00000001’), and possibly a channel name, ‘output’ in the following example.

```
cout << "toto";
[00000002:output] "toto"
```

urbiscript
Session

35.2.6 urbiscript Assertions

The following *assertion frame*:

```
true;
1 < 2;
1 + 2 * 3 == 7;
"foobar"[0, 3] == "foo";
[1, 2, 3].map (function (a) { a * a }) == [1, 4, 9];
[ => ].empty;
```

Assertion
Block

denotes the following assertion-block (see [Section 23.9](#)) in an urbiscript-session frame:

```
assert
{
    true;
    1 < 2;
    1 + 2 * 3 == 7;
    "foobar"[0, 3] == "foo";
    [1, 2, 3].map (function (a) { a * a }) == [1, 4, 9];
    [ => ].empty;
};
```

urbiscript
Session

Chapter 36

Release Notes

This chapter (also known as the Urbi ChangeLog, or Urbi NEWS) lists the user-visible changes in Urbi SDK releases.

36.1 Urbi SDK 2.7.5

Released on 2012-01-27.

This release fixes some packaging-related minor issues.



2012-01-27

36.1.1 Fixes

- The source tarballs are significantly smaller.
- Some public C++ headers used to depend on private ones; this is fixed.
- In-place builds from the source tarballs (i.e., when compiling in the same directory as the source) work. Yet, we still discourage them ([Section 21.4](#)).
- The test suite properly skips tests when some preconditions are not met (e.g., Java support not compiled in, or running as root, or `socat` not being available).
- Windows packages with the installer now have some urbscript files (namely ‘`platform.u`’) which properly depend on whether you are using the debug or release flavor.
- GeSHi support is properly installed.
- When defining functions, their qualified name may start with `this`:

```
function this.foo() { echo("foo"); }|;
this.foo();
[00016170] *** foo
```

urbscript
Session

- Timestamps now use a monotonic clock, insensitive to wall clock changes (fired by `ntpdate` for instance).

36.1.2 Changes

- Compatibility with Clang++ 2.1 and GCC 4.6.
- Compatibility with Boost 1.48. Beware that because of bugs in `Boost.Foreach`¹ (see [Ticket 6131](#)²), Urbi SDK now defines a macro `foreach` much more widely than before. Never include ‘`boost/foreach.hpp`’, rather, use ‘`libport/foreach.hh`’.

¹[Boost.Foreach](http://www.boost.org/doc/libs/release/libs/foreach/), <http://www.boost.org/doc/libs/release/libs/foreach/>.

²[Ticket 6131](https://svn.boost.org/trac/boost/ticket/6131), <https://svn.boost.org/trac/boost/ticket/6131>.

- Thanks to Adam Oleksy, we now provide Debian and RedHat packages.
- `urbi-launch-java` supports the new option ‘[’C]check which checks if Java support is available.
- We now use the version 8 of the Independent JPEG Group’s (IJG) ‘`libjpeg`’.
- To improve performances, the Boolean operators `&&` and `||` can no longer be overridden. In other words, `a && b` no longer maps to `a.’&&’(b)`, but to `if (a) b else a` (with provisions to avoid multiple computations of `a`).
- `System.timeReference` is a `Date`.

36.1.3 New Feature

- `Date` features microsecond support (`Date.microsecond`, `Date.us`).
- `Duration` is accurate at the microsecond.

36.1.4 Documentation

- Extensive overhaul of the HTML rendering of the documentation; compare <http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-sdk.htmldir/> to <http://www.gostai.com/downloads/urbi/2.7.4/doc/urbi-sdk.htmldir/>. Please, pay extra attention to the “Index” button.
- The HTML documentation is also available as a single HTML document (<http://www.gostai.com/downloads/urbi/2.7.5/doc/urbi-sdk-single.htmldir/>).
- Table of figures (Chapter 41) and table of tables (Chapter 40).
- Bibliography, Chapter 38.
- `urbi-launch-java` is documented, Section 22.6.
- The developer documentation for Urbi SDK Remote Java is included in the binary packages (as ‘share/doc/urbi-sdk/doc/sdk-remote-java.htmldir’).

36.2 Urbi SDK 2.7.4

Released on 2011-11-17.

2011-11-17. This release back-ports several fixes from the forthcoming next major release of Urbi, to the 2.7 family.

36.2.1 Fixes

- Freezing an event handler could prevent other event handlers to function properly.
- The indentation of the Emacs urbiscript mode is fixed (contributed by Jeremy W. Sherman).
- Binding a single UVar several times no longer crashes.
- Handling of text and binary files on Windows should no longer be a problem.
- Disconnection of remote UObjects behaves properly.
- In Windows packages, the suffixes of the library (e.g., ‘-vc90-d’) are restored.



36.2.2 Changes

- Boost requirement is now 1.40 instead of 1.38.
- The binary packages for Windows, Mac OS X, Debian Etch are built with Boost 1.47 (from Boost Pro, MacPorts, and install by hand).
- The binary packages for GNU/Linux Ubuntu Lucid are built with the packaged version of Boost: 1.40.
- All the binary packages are now built with ROS Diamondback instead of ROS CTurtle.
- We now provide pkg-config files: ‘libport.pc’ and ‘urbi.pc’. Since binary packages are relocatable, it should be noted that the *prefix* is most probably wrong, so it should be defined at runtime as the output of the urbi’s (and urbi-launch’s) new option ‘print-root’:

```
$ pkg-config urbi --cflags
-I/prefix/include
# But urbi was installed in /usr/local, not in /prefix.
# There is nothing there.
$ ls /prefix/include
ls: /prefix/include: No such file or directory

# Let urbi give urbi-root (or urbi-prefix) to pkg-config:
pkg-config --define-variable=prefix=$(urbi --print-root) urbi --cflags
-I/usr/local/bin/..../include
# This time, it exists.
$ ls /usr/local/bin/..../include
boost jconfig.h jmorecfg.h libport urbi
gostai jerror.h jpeglib.h serialize
```

Shell Session

- GeSHi (Generic Syntax Highlighter)³ support to display colored urbiscript on websites using php.

36.2.3 Documentation

- Instructions to build Urbi SDK are more precise. The requirements should be easier to find ([Section 21.1](#)).
- Formatting of the naming standard is improved ([Chapter 26](#)).
- Instruction for exchanging UObject between UObjects have been clarified ([Section 5.14](#)).
- Various errors in the documentation of UObject were fixed.

36.3 Urbi SDK 2.7.3

Released on 2011-10-07.



2011-10-07.

36.3.1 Fixes

- File descriptor leaks when using `Process`.

³GeSHi (Generic Syntax Highlighter), <http://qbnz.com/highlighter/>.

36.3.2 Changes

- Compatibility with Boost 1.46.
- Binary packages now include simple aliases to the Boost libraries (e.g., you may use ‘-lboost_date_time’ instead of ‘-lboost_date_time-gcc44-mt-1_38’).
- Binary packages on Ubuntu Lucid now use its native Boost libraries (1.40) instead of Boost 1.38, and were built with ROS Diamondback.

36.3.3 Documentation

- Support for Gostai Jazz ([Chapter 30](#)).

36.4 Urbi SDK 2.7.2

Released on 2011-05-13.

2011-05-13. There was no public release of Urbi SDK 2.7.2.

36.4.1 Fixes

- Avoid unexpected execution of code caused by `Semaphore` release and a stop of the code acquiring the semaphore. This caused pieces of code such as the following not to end.

urbiscript
Session

```
var m = Mutex.new();
for& (10)
    { m.stop | m: { sleep(1) } };
"done";
[00016170] "done"
```

Now `Semaphore.acquire` either returns when it holds the semaphore or it jumps to the end of the stopped tag.

36.5 Urbi SDK 2.7.1

Released on 2011-03-17.

2011-03-17.

36.5.1 Fixes

- Crash when stopping a UObject threaded function via a tag.
- On Mac OS X, `umake` and friends pass the ‘-arch’ option. It is now easier to use on a 64 bit computer an Urbi SDK package built on a 32 bit one.

36.5.2 Changes

- The default activation for `GD_CATEGORY` is computed from the first character: ‘`Libport.Path`’ is equivalent to ‘`*,+Libport.Path`’, and ‘`-Urbi*`’ is equivalent to ‘`+*,-Urbi*`’. See [Section 22.1.2](#).
- `System.requireFile` supports the same arguments as `System.load`.

36.5.3 Documentation

- [Chapter 4](#), a quick introduction to some of the basic features of the UObject architecture.
- [uobjects](#).
- gnu.bytecode license, [Section 37.4](#).

36.6 Urbi SDK 2.7



Released on 2011-03-10.

Many optimizations have been implemented, and users should observe a significant speedup. Particularly, the threaded support in UObjects has been modified to perform all operations asynchronously, instead of locking the engine.

2011-03-10.

36.6.1 Changes

- WeakDictionary, WeakPointer are removed. `UVar.notifyAccess`, `UVar.notifyChange`, and `UVar.notifyChangeOwned` no longer need a handler as first argument. Instead of:

```
var myHandle = WeakPointer.new();
&sensorsLoad.notifyChange(myHandle,
    closure() { if (sensorsLoad) sensorsOn else sensorsOff; });
```

urbiscript
Session

write:

```
&sensorsLoad.notifyChange(closure()
    { if (sensorsLoad) sensorsOn else sensorsOff; });
```

urbiscript
Session

Backward compatibility is ensured, but a warning will be issued.

- The functions `urbi::convertRGBtoYCrCb` and `urbi::convertYCrCbtoRGB` have been renamed as `urbi::convertRGBtoYCbCr` and `urbi::convertYCbCrtoRGB` (i.e., a change from ‘YCrCb’ to ‘YCbCr’). Because the previous behavior of these functions was incompatible with their names, after careful evaluation, it was decided not to maintain backward compatibility: it is better to make sure that code that depends on these functions is properly adjusted to their semantics.

36.6.2 New Features

- `Logger` provides a logging service:

```
var logger = Logger.new("Category")|;

logger.dump << "Low level debug message"|;
// Nothing displayed, unless the debug level is set to DUMP.

logger.warn << "something wrong happened, proceeding"|;
[     Category      ] something wrong happened, proceeding

logger.err << "something really bad happened!"|;
[     Category      ] something really bad happened!
```

urbiscript
Session

- `Profile` and `Profile.Function` replace the former `Profiling` object.
- `Stream`, common prototype to `InputStream` and `OutputStream`.
- `System.sleep`’s argument now defaults to `Float.inf`.
- Controlling the maximum queue size for UObject threaded notifies and bound functions is now possible, see [Section 5.4.3](#).
- `at` now comes in synchronous and asynchronous flavors, see [Section 23.10.1.3](#).
- A new construct, `watch`, creates an event that allows to monitor any change of an expression ([Section 23.10.3](#)).

urbiscript
Session

```

var x = 0|;
var y = 0|;
var e = watch(x + y)|;
at (e?(var value))
    echo("x + y = %s" % value);
x = 1|;
[00000000] *** x + y = 1
y = 2|;
[00000000] *** x + y = 3

```

- Gostai Editor for Windows was updated to 2.5. It now includes advanced search and replace features with regular expression support and a “find in all opened documents” option. “Goto line” menu has also been added.
- Gostai Console 2.6 for Windows now offers autocompletion of urbiscript slot names.
- To make simpler to install several versions of Urbi SDK, the Windows installers now include the version number in the destination path.

36.6.3 Documentation

- [Chapter 19](#), A programming guideline in Urbi SDK.
- [UVar.notifyChangeOwned](#), [UVar.removeNotifyAccess](#), [UVar.removeNotifyChange](#), and [UVar.removeNotifyChangeOwned](#).
- [urbi-sound](#) ([Section 22.9](#)).



36.7 Urbi SDK 2.6

Released on 2011-01-06.

2011-01-06.

This release features several deep changes that are not user visible, but which provide significant optimizations. Several bugs have been fixed too.

36.7.1 Fixes

- Improper behavior when there are several concurrent `at (exp ~ duration)`.
- System interruptions with Control-C sometimes failed.
- Remote UObjects exit properly when the server shuts down.
- Binary packages provide RTP support for all the architectures.

36.7.2 Optimizations

- urbiscript interpretation has been globally sped up by around 30%.
- Event emission routines have been optimized: `Event.'emit'` by around 25%, `Event.syncEmit` by around 13%, `Event.trigger` by around 25% and `Event.syncTrigger` by around 10%.

36.7.3 New Features

- The usual C++ syntax to declare classes with multiple inheritance is supported, see [Section 23.1.6.8](#).
- A literal syntax for strict variadic functions has been added, see [Section 23.3.7](#). For instance

```
function variadic(var a1, var a2, var a3, var args[])
{
    echo("a1 = %s, a2 = %s, a3 = %s, args = %s"
        % [a1, a2, a3, args]);
}
variadic(1, 2, 3);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = []
variadic(1, 2, 3, 4);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = [4]
variadic(1, 2, 3, 4, 5);
[00000002] *** a1 = 1, a2 = 2, a3 = 3, args = [4, 5]
```

urbiscript
Session

This is faster than using lazy functions and call messages.

- `Directory` objects have new features for creation, modification and deletion.

```
Directory.createAll("dir1/dir2/dir3")|;
Directory.new("dir1").rename("dir")|;
Directory.new("dir/dir2").copy("dir/dir4")|;
Directory.new("dir").removeAll;
```

urbiscript
Session

- `Directory.size`, `File.size`, `Directory.lastModifiedDate`, `File.lastModifiedDate`.

36.7.4 Documentation

- `Lobby.bytesReceived`, `Lobby.bytesSent`.

36.8 Urbi SDK 2.5

Released on 2010-12-07.



2010-12-07

36.8.1 Fixes

- Memory consumption at start-up is reduced.

36.8.2 New Features

- `urbi-launch` and `urbi-send` support `'-m'`/`--module=file`, to load a module ([Section 22.5](#), [Section 22.8](#)).
- The search paths for urbiscript files and for UObject files can be changed from urbiscript (`System.searchPath`, `UObject.searchPath`).
- New syntactic sugar for `Object.getSlot`: `o.&name` is equivalent to `o.getSlot("name")` (and `&name` is equivalent to `getSlot("name")`). For instance, instead of

```
function Derive.init(var arg)
{
    Base.getSlot("init").apply([this, arg]);
};

function Foo.'=='(var that)
{
```

urbiscript
Session

```
    getSlot("accessor") == that.getSlot("accessor");
};
```

write

urbiscript
Session

```
function Derive.init(var arg)
{
  Base.&init.apply([this, arg]);
};

function Foo.'=='(var that)
{
  &accessor == that.&accessor;
};
```

- [Dictionary](#) keys can now be arbitrary objects. Objects hashing can be overridden. See [Object.hash](#).
- [Global.warn](#) sends messages prefixed with !!! (as error messages), instead of ***.
- [Hash](#), type for hash codes for [Dictionary](#).
- [List.insertUnique](#) inserts a member if it's not already part of the list.
- [Object.hash](#), [Float.hash](#), [String.hash](#), [List.hash](#).
- [Object.removeLocalSlot](#). It raises an error when asked to remove of a non-existing slot. Please note that, contrary to what its name suggests, [Object.removeSlot](#) is only removing *local* slots. Using [Object.removeLocalSlot](#) is encouraged.
- [System.eval](#), [System.load](#), and [System.loadFile](#) accept an optional second argument, the context ([this](#)) of the evaluation.

36.8.3 Changes

- More types of empty statements are warned about. For instance Urbi used to accept silently `if (foo);`. It now warns about the empty body, and recommends `if (foo) {};`.
- [Dictionary.erase](#) raises an error if the key does not exist.
- [Exception.ArgumentType](#) is deprecated, use [Exception.Argument](#) that wraps around any [Exception](#) instead.
- [Object.getProperty](#) raises an error if the property does not exist. It used to return `void`.
- [Object.removeSlot](#) warns when asked to remove of a non-existing slot, and [Object.removeSlot](#) about non-existing properties.

urbiscript
Session

```
removeSlot("doesNotExist")|;
[00000002:warning] !!! no such local slot: doesNotExist
[00000002:warning] !!!      called from: removeSlot
```

In the past, it used to accept this silently; in the future, this will be an error, as with [Object.removeLocalSlot](#). Use [Object.hasLocalSlot](#) or [Object.hasProperty](#) beforehand, if needed.

- A warning is now issued when the evaluation of the condition of an [at](#) statement yields an Event and no question mark was used, since this is most likely an oversight.

urbiscript
Session

```

var e = Event.new;
[00000001] Event_OxADDR

// This is not the correct way to match an event.
at (e) echo("Oops.");
[00000002:warning] !!! at (<event>) without a '?', this is probably not what you meant.
[00000003] *** Oops.

// This is.
at (e?) echo("Okay.");

```

- `File.rename` returns `this`.

36.8.4 Documentation

- The `urbi-ping` program, [Section 22.7](#).
- Properties, [Section 12.7](#).
- The `class` statement is better described in [Section 12.4](#) and [Section 23.1.6.8](#).
- The tutorial documents the payloads in event-based constructs ([Section 15.2.2](#)).

36.9 Urbi SDK 2.4

Released on 2010-10-20.



2010-10-20.

36.9.1 Fixes

- Fix transmission of binary data in dictionaries from remote UObjects.
- Fix sound conversion between mono and stereo.
- Fix the `urbi-sendsound` liburbi example.
- Fix liburbi stream formatting that made the code waste a lot of bandwidth.
- Optimize runtime performances when log messages are inhibited.
- Preserve properties when a property assignment triggers a copy-on-write.
- Trigger the `changed` attribute of tags when they are frozen, unfrozen, blocked, ...
This fixes `at (myTag.frozen) ...;`
- Fix `Float.random` on windows, which always returned 41 in some situations.
- Do not scope variables created inside a pipe inside a comma:

```

var a = 0 | var b = 1,
echo(a);
[00000001] *** 0
echo(b);
[00000001] *** 1

```

urbiscript
Session

36.9.2 New Features

- Issue a warning when an UObject is plugged in a different version of the kernel than the SDK that compiled it — which can provoke undefined runtime behavior.
- Java API for UObject, see [Chapter 6](#).
- Enumerations.

Enumeration types can be created with the usual C-like syntax. See [Section 23.5](#) and [Enumeration](#).

urbiscript
Session

```
enum Suit
{
    hearts,
    diamonds,
    clubs,
    spades, // Last comma is optional
};

[00000001] Suit

for (var suit : Suit)
    if (suit in [Suit.spades, Suit.clubs])
        echo("Black: " + suit)
    else
        echo("Red: " + suit);
[00000001] *** Red: hearts
[00000002] *** Red: diamonds
[00000003] *** Black: clubs
[00000004] *** Black: spades
```

- `umake` supports new options: ‘-I’, ‘-L’, ‘-l’, ‘--package’ (for `pkg-config`). See [Section 22.10](#). The documentation of `umake` now describes `EXTRA_CPPFLAGS`, `EXTRA_CXXFLAGS`, and `EXTRA_LDFLAGS`.
- RTP mode can now be switched on at any time. Change is applied to existing notifies.
- UObjects can now be instantiated directly from C++, both in plugin and remote mode.
- New mechanism to map simple structures between C++ and urbiscript ([Section 5.18](#)).
- `nil` is now correctly serialized to/from remote UObject.
- The C++ header ‘`urbi/revision.hh`’ contains version information that can be used to set requirements. For instance:

C++

```
#include <urbi/revision.hh>
#if URBI_SDK_VERSION_VALUE < 2003000
# error Urbi SDK 2.3 or better is required.
#endif
```

- `Date.year`, `Date.month`, `Date.day`, `Date.hour`, `Date.minute`, `Date.second`.
- The keys in Dictionary literals are no longer required to be literal strings.

urbiscript
Session

```
["a" + "b" => "ab", 12.asString => "12"];
[00002405] ["12" => "12", "ab" => "ab"]
```

They still need to evaluate into String values. Note: this is no longer true since Urbi SDK 2.5.

urbiscript
Session

```
[12 => "12"];
[00005064:error] !!! unexpected 12, expected a String
```

36.9.3 Documentation

- `String.empty`, `String.length`.

36.10 Urbi SDK 2.3

Released on 2010-09-28.



2010-09-28.

36.10.1 Fixes

- `Date.asFloat` is restored.
- `File.create` empties existing files first.
- `Lobby.lobby` always returns the current lobby, even if invoked on another lobby.
- `Object.inspect` works properly, even if the target is a remote lobby.
- `Regexp.matchs` renamed as `Regexp.matches`.
- `System.version` Really returns the current version.
- Fix multiple race conditions in RTP handling code preventing proper initialization of remote UObjects.
- Fix Windows deployment to have both debug and release UObjects installed.
- Fix `urbi-sound` in the liburbi examples.
- Fix server mode of ‘`urbi-launch --remote`’.

36.10.2 New Features

- The documentation of Urbi SDK Remote, our middleware layer to communicate with an Urbi server — either by hand or via the UObjects —, is included in the binary packages (in ‘`share/doc/urbi-sdk/sdk-remote.htmldir/index.html`’. It is also available on-line at <http://www.gostai.com/downloads/urbi/doc/sdk-remote.htmldir>.
- In addition to Gostai Console 2.5, Windows installers of Urbi SDK now include the Gostai Editor 2.4.1.
- By popular demand, all the Boost libraries (1.38) are included in the binary packages. We used to provide only the headers and libraries Urbi SDK depends upon. Boost.Python, because it has too many dependencies, is not included.
- When launched with no input (i.e., none of the options ‘`-e`’/‘`--expression`’, ‘`-f`’/‘`--file`’, ‘`-P`’/‘`--port`’ were given), the interpreter is interactive.
- Assignment operators such as ‘`+ =`’ are redefinable. See [Section 23.1.8.2](#).
- `Date.'-'` accepts a `Duration` or a `Float` in addition to accepting a `Date`.
- `Date.year`, `Date.month`, `Date.day`, `Date.hour`, `Date.minute`, `Date.second` slots allow partial modifications of `Date` objects.
- `Float.fresh` generates unique integers.
- `InputStream.close`.
- `List.'+ ='`.

- Support for `else` in `try` blocks (Section 23.8.2). Run only when the `try` block completed properly.

urbscript
Session

```
// Can we run "riskyFeature"?
try { riskyFeature } catch { false } else { true };
[00004220] false

function riskyFeature() { throw "die" }|;
try { riskyFeature } catch { false } else { true };
[00004433] false

riskyFeature = function () { 42 }|;
try { riskyFeature } catch { false } else { true };
[00004447] true
```

- Support for `finally` in `try` blocks (Section 23.8.4). Use it for code that must be run whatever the control flow can be. For instance:

urbscript
Session

```
try { echo(1) } catch { echo(2) } else { echo(3) } finally { echo(4) };
[00002670] *** 1
[00002670] *** 3
[00002670] *** 4

try { throw 1 } catch { echo(2) } else { echo(3) } finally { echo(4) };
[00002671] *** 2
[00002671] *** 4
```

- `System.eval` and `System.load` report syntax warnings.

urbscript
Session

```
eval("new Object");
[00001388:warning] !!! 1.1-10: 'new Obj(x)' is deprecated, use 'Obj.new(x)'
[00001388:warning] !!!      called from: eval
[00001388] Object_0x1001b2320
```

- New functions `as` and `fill` on `UVar` to ease access to the generic cast system.
- Add support to `boost::unordered_map` to `UObject` casting system.
- Optimize remote `UObjects`: notifies between two objects in the same process instance are transmitted locally.
- Provide a `CustomUVar` class to ease encapsulation of custom data in `UVar`.
- Bind the `constant` property on `UVar`.

36.11 Urbi SDK 2.2

Released on 2010-08-23.

2010-08-23.

36.11.1 Fixes

- The main loop optimization triggered several issues with GNU/Linux and Mac OS X in interactive sessions (truncated output, possibly blocked input). These issues are fixed on Snow Leopard and Leopard and GNU/Linux.
- Deep overall of the event handling primitives. Watching an expression will succeed in cases where it used to fail. For instance:

urbscript
Session

```

at (isdef (myVar))
echo("var myVar = " + myVar),
myVar;
[00000001:error] !!! lookup failed: myVar

var myVar = 42|;
[00000003] *** var myVar = 42

```

or

```

var x = 0|;
function f() { x == 1 }|;
at (f()) echo("OK");
x = 1|;
[00000001] *** OK

```

urbiscript
Session

Support for sustained events is fixed too (see [Section 23.10.1](#)).

- `List.max` and `List.min` will now report the right indexes in error messages.
- `onleave` blocks on lasting events are now run asynchronously, as expected.
- `at (expression ~ duration)` is now supported.
- Fix a bug if emitting an event triggers its unsubscription.
- Fix printing of `Exception.Type` and `Exception.ArgumentType`.
- Fix timestamp overflow on Windows after 40 minutes.
- Fix fatal error when manipulating the first `Job` prototype.

36.11.2 New Features

- Pressing C-c in the urbiscript shell ('`urbi -i`') interrupts the foreground job, and clears the pending commands. A second C-c in a row invokes `System.shutdown`, giving a chance to the system to shut down properly. A third C-c kills `urbi/urbi-launch`. See [Section 22.3.2](#) for more details.
- Closing the standard input (e.g., by pressing C-d) in interactive sessions shuts down the server.
- Remote UObjects now support the RTP protocol to exchange value with the engine ([Section 5.17](#)).
- NotifyChange/access callbacks can now take any type as argument. `UVar&` still has the previous behavior. For any other type, the system will try to convert the value within the `UVar` to this type.
- `CallMessage.eval`.
- `Float.ceil`, `Float.floor`, `Float.isInf`, `Float.isnan`.
- `Traceable`.
- Improved context (the call stacks) when errors are reported. Especially when using `System.eval` or `System.load`.

- `at` (*expression*) — as opposed to `at` (*event?*) — implementation has been improved: the condition will now be reevaluated even if a parameter not directly in the expression (in the body of a called function, for instance) is modified.
- `Regexp.matchs`.
- `Date` objects now have microsecond resolution and have been slightly revamped to not rely on Unix's epoch.
- UVars now have the timestamp of their latest assignment.

36.11.3 Documentation

- `Float.hex`.

36.12 Urbi SDK 2.1

Released on 2010-07-08.

2010-07-08.

36.12.1 Fixes

- `Lobby.connectionTag` monitors the jobs launched from the lobby, but can no longer kill the lobby itself.
- ‘123foo’ is no longer accepted as a synonym to ‘123 foo’. As a consequence, in case you were using `x = 123cos: 1`, convert it to `x = 123 cos: 1`.
- Some old tools that no longer make sense in Urbi SDK 2.0 have been removed: `umake-engine`, `umake-fullengine`, `umake-lib`, `umake-remote`. Instead, use `umake`, see [Section 22.10](#).
- On Windows `urbi-launch` could possibly miss module files to load if the extension (‘.dll’) was not specified. One may now safely, run ‘`urbi-launch my-module`’ (instead of ‘`urbi-launch my-module.dll`’ or ‘`urbi-launch my-module.so`’) on all the platforms.

36.12.2 New Features

- `Regexp.asPrintable`, `Regexp.asString`, `Regexp.has`.
- `System.Platform.host`, `System.Platform.hostAlias`, `System.Platform.hostCpu`, `System.Platform.hostOs`, `System.Platform.hostVendor`.
- UObject init method and methods bound by `notifyChange` no longer need to return an int.
- `Channel.Filter`, a `Channel` that outputs text that can be parsed without error by the liburbi.
- `RangeIterable.all`, `RangeIterable.any`, moved from `List`.
- Support for `ROS`, the Robot Operating System. See [Chapter 16](#) for an introduction, and [Chapter 25](#) for the details.
- `Lobby.lobby` and `Lobby.instances`, bounced to from `System.lobby` and `System.lobbies`.
- `Tag.scope`, bounced to from `System.scopeTag`.

36.12.3 Optimization

- The main loop was reorganized to factor all socket polling in a single place: latency of `Socket` is greatly reduced.

36.12.4 Documentation

- `Lobby.authors`, `Lobby.thanks`.
- `System.PackageInfo`.
- `System.spawn`.
- LEGO Mindstorms NXT support ([Chapter 29](#)).
- Pioneer 3-DX support ([Chapter 31](#)).
- Support for Segway RMP ([Chapter 32](#)).

36.13 Urbi SDK 2.0.3

Released on 2010-05-28.



2010-05-28.

36.13.1 New Features

- `Container`, prototype for `Dictionary`, `List` derive.
- `e not in c` is mapped onto `c.hasNot(e)` instead of `!c.has(e)`.
- `Float.limits`
- `Job.asString`
- `IoService`
- `Event.'<<'`
- `List.argmax`, `List.argmin`, `List.zip`
- `Tuple.'+'`
- `Tuple.*'`
- Assertion failures are more legible:

```
urbiscript
Session
var one = 1|;
var two = 2|;
assert (one == two);
[00000002:error] !!! failed assertion: one == two (1 != 2)
```

instead of

```
urbiscript
Session
assert (one == two);
[00000002:error] !!! failed assertion: one.'=='(two)
```

previously. As a consequence, `System.assert_op` is deprecated. The never documented following slots have been removed from `System`: `assert_eq`, `assert_ge`, `assert_gt`, `assert_le`, `assert_lt`, `assert_meq`, `assert_mne`, `assert_ne`.

36.13.2 Fixes

- `List.<` and `Tuple.<` implement true lexicographic order: `[0, 4] < [1, 3]` is true. List comparison used to implement member-wise comparison; the previous assertion was not verified because `4 < 3` is not true.
- `Mutex.asMutex` is fixed.
- `Directory` events were not launched if a `Directory` had already been created on the same `Path`.
- `waituntil` no longer ignores pattern guards.

36.13.3 Documentation

- Bioloid ([Chapter 28](#)).
- Garbage collection ([Section 23.12](#)).
- Structural Pattern matching ([Section 23.6](#)).
- `CallMessage.sender` and `CallMessage.target`.
- `Dictionary.asString`.
- `Directory.fileCreated` and `Directory.fileDeleted`.
- `List.max`, `List.min`.
- `Mutex.asMutex`.
- `Object.localSlotNames`.

36.14 Urbi SDK 2.0.2

Released on 2010-05-06.

2010-05-06.

36.14.1 urbiscript

- `Control.detach` and `Control.disown` return the `Job`.

36.14.2 Fixes

- ‘`make install`’ failures are addressed.
- `freezeif` can be used more than once inside a scope.

36.14.3 Documentation

- `StackFrame`
- `String.split`

36.15 Urbi SDK 2.0.1

Released on 2010-05-03.

2010-05-03.

36.15.1 urbiscript

- Minor bug fixes.
- The short option ‘-v’ is reserved for ‘--verbose’. Tools that mistakenly used ‘-V’ for ‘--verbose’ and ‘-v’ for ‘--version’ have been corrected (short options are swapped). Use long options in scripts, not short options.
- `Lobby.echoEach`: new.
- `String.closest`: new.
- `Tuple.size`: new.

36.15.2 Documentation

- How to build Urbi SDK ([Chapter 21](#)).
- Hyperlinks to slots (e.g., `Float.asString`).

36.15.3 Fixes

- Closures enclose the lobby. Now slots of the lobby in which the closure has been defined are visible in functions called from the closure.

36.16 Urbi SDK 2.0

Released on 2010-04-09.



2010-04-09.

36.16.1 urbiscript

36.16.1.1 Changes

- `Global.Tags` is renamed as `Tag.tags`.
- `Global.Task` is renamed as `Global.Job`.
- `Global.topLevel` is renamed as `Channel.topLevel`.
- `Global.output`, `Global.error` are removed, they were deprecated in favor of `Global.out` and `Global.err`.
- `Object.getPeriod` is deprecated in favor of `System.period`.
- As announced long ago, and as displayed by warnings, `Object.slotNames` now returns all the slot names, ancestors included. Use `Object.localSlotNames` to get the list of the names of the slot the object owns.
- `Semaphore.acquire` and `Semaphore.release` are promoted over `Semaphore.p` and `Semaphore.v`.

36.16.1.2 New features

- Dictionary can now be created with literals.

Syntax	Semantics
<code>[=>]</code> <code>["a" => 1, "b" => 2, "c" => 3]</code>	<code>Dictionary.new</code> <code>Dictionary.new("a", 1, "b", 2, "c", 3)</code>

- `Float.random`
- `List.subset`
- `Object.getLocalSlot.`
- String escapes accept one- and two-digit octal numbers. For instance "`\0`", "`\00`" and "`\000`" all denote the same value.
- Tuple can now be created with literals.

Syntax	Semantics
<code>()</code>	<code>Tuple.new([])</code>
<code>(1,)</code>	<code>Tuple.new([1])</code>
<code>(1, 2, 3)</code>	<code>Tuple.new([1, 2, 3])</code>

- Location.'=='.
- type replaces '\$type'

36.16.2 UObjects

- Remote timers (USetUpdate, USetTimer) are now handled locally instead of by the kernel.
- UVars can be copied using the `UVar.copy` method.
- New UEvent class, similar to UVar. Can be used to emit events.
- Added support for dictionaries: new UDictionary structure in the UValue union.

36.16.3 Documentation

- `Barrier`
- `Date.now`
- `Float.random`
- `Pattern`
- `PubSub`
- `PubSub.Subscriber`
- `Profiling`
- `Semaphore`
- Trajectories
- `TrajectoryGenerator`
- `urbi-image` ([Section 22.4](#)).
- `waituntil` clauses
- `whenever` clauses

36.17 Urbi SDK 2.0 RC 4

Released on 2010-01-29.



2010-01-29.

36.17.1 urbiscript

36.17.1.1 Changes

- '\$id' replaces id
- List derives from Orderable.

36.17.1.2 New objects

- [Location](#)
- [Position](#)

36.17.1.3 New features

- [File.remove](#)
- [File.rename](#)

36.17.2 UObjects

- The UObject API is now thread-safe: All UVar and UObject operations can be performed from any thread.
- You can request bound functions to be executed asynchronously in a different thread by using UBindThreadedFunction instead of UBindFunction.

36.18 Urbi SDK 2.0 RC 3

Released on 2010-01-13.



2010-01-13.

36.18.1 urbiscript

36.18.1.1 Fixes

- 'local.u' works as expected.

36.18.1.2 Changes

- [Lobby.quit](#) replaces [System.quit](#).
- [Socket.connect](#) accepts integers.
- UObject remote notifyChange on USensor variable now works as expected.
- UObject timers can now be removed with `UObject::removeTimer()`.

36.18.2 Documentation

- [Socket](#) provides a complete example.
- The Naming Standard documents the support classes provided to ease creation of the component hierarchy.



36.19 Urbi SDK 2.0 RC 2

Released on 2009-11-30.

2009-11-30.

This release candidate includes many fixes and improvements that are not reported below. The following list is by no means exhaustive.

36.19.1 Optimization

The urbiscript engine was considerably optimized in both space and time.

36.19.2 urbiscript

36.19.2.1 New constructs

- `assert` { claim1; claim2;... };
- `every|`
- `break` and `continue` are supported in `every|` loops.
- `for(num)` and `for(var i: set)` support the `for&`, `for|` and `for;` flavors.
- `for(init; cond; inc)` supports the `for|` and `for;` flavors.
- non-empty lists of expressions in list literals, in function calls, and non-empty lists of function formal arguments may end with a trailing optional comma. For instance:

urbiscript
Session

```
function binList(a, b,) { [a, b,] } | binList(1, 2,)
```

is equivalent to

urbiscript
Session

```
function binList(a, b) { [a, b] } | binList(1, 2)
```

- consecutive string literals are joined into a unique string literal, as in C++.

36.19.2.2 New objects

- `Component`, `Localizer`, `Interface`: naming standard infrastructure classes.
- `Date`
- `Directory`
- `File`
- `Finalizable`: objects that call `finalize()` when destroyed.
- `InputStream`
- `Mutex`
- `OutputStream`
- `Process`: Start and monitor child processes.
- `Regexp`
- `Server`: TCP/UDP server socket.
- `Socket`: TCP/UDP client socket.
- `Timeout`
- `WeakDictionary`, `WeakPointer`: Store dictionary of objects without increasing their reference count.

36.19.2.3 New features

- `Object.asBool`.
- `Lobby.wall`.
- `Dictionary.size`.
- `Global.evaluate`.
- `Group.each`, `Group.each&`
- `Lobby.onDisconnect`, `Lobby.remoteIP` `Lobby.create`.
- `Object.inspect`.
- `String.fromAscii`, `String.replace`, `String.toAscii`.
- System: `_exit`, `assert_eq`, `System.system`.

36.19.2.4 Fixes

- at constructs do not leak local variables anymore.
- Each tag now has its enter and leave events.
- `File.content` reads the whole file.
- Invalid assignments such as `f(x) = n` are now refused as expected.

36.19.2.5 Deprecations

- `Object.ownsSlot` is deprecated in favor of `Object.hasSlot`/`Object.hasLocalSlot`.
- `Object.slotNames` is deprecated in favor of `Object.allSlotNames`/`Object.localSlotNames`.

36.19.2.6 Changes

- empty strings, dictionaries and lists are now evaluated as `false` in conditions.
- `Dictionary.asString` does not sort the keys.
- `Dictionary.'[]='` returns the assigned value, not the dictionary.
- `Dictionary.'[]'` raises an exception if the key is missing.
- Constants is merged into Math.
- `every` no longer goes in background. Instead of:

```
every (1s) echo("foo");
```

urbiscript
Session

write (note the change in the separator)

```
every (1s) echo("foo"),
```

urbiscript
Session

or

```
detach({ every (1s) echo("foo"); });
```

urbiscript
Session

- Tag: begin and end now simply print the tag name followed by ‘begin’ or ‘end’.
- System-code is now hidden from the backtraces.
- `Code.apply`: the call message can be changed by passing it as an extra argument.

36.19.3 UObjects

- Handle UObject destruction. To remove an UObject, call the urbiscript `destroy` method. The corresponding C++ instance will be deleted.
- Add `UVar::unnotify()`. When called, it removes all `UNotifyChange` registered with the `UVar`.
- Bind functions using `UBindFunction` can now take arguments of type `UVar&` and `UObject*`. The recommended method to pass UVars from urbiscript is now to use `camera.getSlot("val")` instead of `camera.val`.
- Add a 0-copy mode for UVars: If '`UVar::enableBypass(true)`' is called on an `UVar`, `notifyChange` on this `UVar` can recover the not-copied data by using `UVar.get()`, returning an `UValue&`. However, the data is only accessible from within `notifyChange`: reading the `UVar` directly will return `nil`.
- Add support for the `changed!` event on UVars. Code like:

urbiscript
Session

```
at (headTouch.val->changed? if headTouch.val)
  tts.say("ouch");
```

will now work. This hook costs one `at` per `UVar`; set `UVar.hookChanged` to false to disable it.

- Add a statistics-gathering tool. Enable it using `uobjects.enableStats`. Reset counters by calling `uobjects.clearStats`. `uobjects.getStats` will return a dictionary of all bound C++ function called, including timer callbacks, along with the average, min, max call durations, and the number of calls.
- When code registered by a `notifyChange` throws, the exception is intercepted to protect other unrelated callbacks. The throwing callback gets removed from the callback list, unless the `removeThrowingCallbacks` on the `UVar` is false.
- the environment variable `URBI_UOBJECT_PATH` is used by urbi-launch and urbiscript's load-Module to find uobjects.
- fixed multiple notifications of event trigger in remote UObject.
- Many other bug fixes and performance improvements.
- an exception is now thrown if the C++ init method failed.

36.19.4 Documentation

The documentation was fixed, completed, and extended. Its layout was also improved. Changes include, but are not limited to:

- various programs: `urbi`, `urbi-launch`, `urbi-send` etc. ([Chapter 22](#)).
- environment variables: `URBI_UOBJECT_PATH`, `URBI_PATH`, `URBI_ROOT` ([Section 22.1](#)).
- special files ‘`global.u`’, ‘`local.u`’ ([Section 22.2](#)).
- k1-to-k2: Conversion idioms from urbiscript 1 to urbiscript 2 ([Chapter 20](#)).
- FAQ ([Chapter 18](#))
 - stack exhaustion
 - at and waituntil: performance considerations

- Specifications:
 - completion of the definition of the control flow constructs (`every`, `everyl`, `if`, `for`, `loop`)
 - tools (umake, umake-shared, umake-deepclean, urbi, urbi-launch, urbi-send).
 - `Boolean`
 - `Channel`
 - `Date`
 - `Dictionary`
 - `Exception`
 - `File`
 - `Kernel1`
 - `InputStream`
 - `Lazy`
 - `Math`
 - `Mutex`
 - `Regexp`
 - `Object`
 - `OutputStream`
 - `Pair`
 - `String`
 - `Tag`
 - `Timeout`
- tutorial:
 - uobjects

36.19.5 Various

- Text files are converted to DOS end-of-lines for Windows packages.
- `urbi-send` supports ‘`--quit`’.
- The files ‘`global.u`’/‘`local.u`’ replace ‘`URBI.INI`’/‘`CLIENT.INI`’.
- `urbi` supports ‘`--quiet`’ to inhibit the banner.

36.20 Urbi SDK 2.0 RC 1

Released on 2009-04-03.



2009-04-03

36.20.1 Auxiliary programs

- `urbi-send` no longer displays the server version banner, unless given ‘`-b`’/‘`--banner`’.
- `urbi-console` is now called simply `urbi`.
- `urbi.bat` should now work out of the box under windows.

36.20.2 urbiscript

36.20.2.1 Changes

- The keyword `emit` is deprecated in favor of `!: instead of emit e(a);, write e!(a);`. The `? construct is changed for symmetry: instead of at (?e(var a)), write at (e?(var a))`. See [Section 20.3](#) for details. This syntax for sending and receiving is traditional and can be found in various programming languages.
- `System.Platform` enhances former `System.platform`. Use `System.Platform.kind` instead of `System.platform`.

36.20.2.2 Fixes

- Under some circumstances successful runs could report “at job handler exited with exception `TerminateException`”. This is fixed.
- Using `waituntil` on an event with no payload (i.e., `waituntil(e?) ...;`) will not cause an internal error anymore.

36.20.3 URBI Remote SDK

The API for plugged-in UObjects is not thread safe, and never was: calls to the API must be done only in the very same thread that runs the Urbi code. Assertions (run-time failures) are now triggered for invalid calls.

36.20.4 Documentation

Extended documentation on: [Comparable](#), [Orderable](#).

36.21 Urbi SDK 2.0 beta 4

Released on 2009-03-03.

2009-03-03.

36.21.1 Documentation

An initial sketch of documentation (a tutorial, and the language and library specifications) is included.

36.21.2 urbiscript

36.21.2.1 Bug fixes

- Bitwise operations.
The native `long unsigned int` type is now used for all the bitwise operations (`&`, `|`, `^`, `compl`, `<<`, `>>`). As a consequence it is now an error to pass negative operands to these operations.
- `System.PackageInfo`.
This new object provides version information about the Urbi package. It is also used to ensure that the initialization process uses matching Urbi and C++ files. This should prevent accidental mismatches due to incomplete installation processes.
- Precedence of operator `**`.
In conformance with the usage in mathematics, the operator `**` now has a stronger precedence than the unary operators. Therefore, as in Perl, Python and others, `'-2 ** 2 == -4'` whereas it used to be equal to `'4'` before (as with GNU bc).



- `whenever` now properly executes the else branch when the condition is false. It used to wait for the condition to be verified at least once before.

36.21.2.2 Changes

- `String.asFloat`.

This new method has been introduced to transform a string to a float. It raises a PrimitiveError exception if the conversion fails:

```
"2.1".asFloat;
[00000002] 2.1
"2.0a".asFloat;
[00000003:error] !!! asFloat: cannot convert to float: "2.0a"
```

urbiscript
Session

36.21.3 Programs

36.21.3.1 Environment variables

The environment variable `URBI_ROOT` denotes the directory which is the root of the tree into which Urbi was installed. It corresponds to the “prefix” in GNU Autoconf parlance, and defaults to ‘`/usr/local`’ under Unix. urbiscript library files are expected to be in `URBI_ROOT/share/gostai/urbi`.

The environment variable `URBI_PATH`, which allows to specify a colon-separated list of directories into which urbiscript files are looked-up, may extend or override `URBI_ROOT`. Any superfluous colon denotes the place where the `URBI_ROOT` path is taken into account.

36.21.3.2 Scripting

To enable writing (batch) scripts seamlessly in Urbi, `urbi-console` ‘`-f`’/‘`--fast`’ is now renamed as ‘`-F`’/‘`--fast`’. Please, never use short options in batch programs, as they are likely to change.

Two new option pairs, ‘`-e`’/‘`--expression`’ and ‘`-f`’/‘`--file`’, plus the ability to reach the command line arguments from Urbi make it possible to write simple batch Urbi programs. For instance:

```
$ cat demo
#! /usr/bin/env urbi-console
cout << System.arguments;
shutdown;

$ ./demo 1 2 3 | grep output
[00000004:output] ["1", "2", "3"]
```

Shell
Session

36.21.3.3 urbi-console

`urbi-console` is now a simple wrapper around `urbi-launch`. Running

```
urbi-console arg1 arg2...
```

Shell
Session

is equivalent to running

```
urbi-launch --start -- arg1 arg2...
```

Shell
Session

36.21.3.4 Auxiliary programs

The command line interface of `urbi-sendbin` has been updated. `urbi-send` now supports ‘`-e`’/‘`--expression`’ and ‘`-f`’/‘`--file`’. For instance

```
$ urbi-send -e 'var x;' -e "x = $value;" -e 'shutdown;'
```

urbiscript
Session

36.22 Urbi SDK 2.0 beta 3

Released on 2009-01-05.

2009-01-05. 36.22.1 Documentation

A new document, ‘FAQ.txt’, addresses the questions most frequently asked by our users during the beta-test period.

36.22.2 urbscript

36.22.2.1 Fixes

- If a file loaded from ‘URBI.INI’ cannot be found, it is now properly reported.

36.22.2.2 Changes

- `new` syntax revamped.

The syntax `new myObject(myArgs)` has been deprecated and now gives a warning. The recommended `myObject.new(myArgs)` is suggested.

- `delete` has been removed.

`delete` was never the right thing to do. A local variable should not be deleted, and a slot can be removed using `Object.removeSlot`. The construct `delete object` has been removed from the language.

- `__HERE__`.

The new `__HERE__` pseudo-symbol gives the current position. It features three self explanatory slots: `file`, `line`, and `column`.

- Operator `()`.

It is now possible to define the `()` operator on objects and have it called as soon as at least one parameter is given:

```
urbascript
Session
class A {
    function '()' (x) { echo("A called with " + x) };
}!;
A;
[00000001] A
A();
[00000002] A
A(42);
[00000003] *** A called with 42
```

- `catch (type name)` syntax removed.

It was used to catch exceptions if and only if they inherited `type`. This behavior can be obtained with the more general guard system:

```
urbascript
Session
catch (var e if e.isA(<type>))
{
    ...
}
```

- Pattern matching and guards in catch blocks.

Exception can now be filtered thanks to pattern matching, just like events. Moreover, the pattern can be followed by the `if` keyword and an arbitrary guard. The block will catch the exception only if the guard is true.

```
try
{ ... }
catch ("foo") // Catch only the "foo" string
{ ... }
catch (var x if x.isA(Float) && x > 10) // Catch all floats greater than 10
{ ... }
catch (var e) // Catch any other exception
{ ... }
```

- Parsing of integer literals.

The parser could not read integer literals greater than $2^{31} - 1$. This constraint has been alleviated; integer literals up to $2^{63} - 1$ are accepted.

- Display of integer literals.

Some large floating point values could not be displayed correctly at the top level of the interpreter. This limitation has been removed.

- Variables binding in event matching.

Parentheses around variables bindings (`var x`) are no longer required in event matching:

```
at (?myEvent(var x, var y, 1))
```

urbiscript
Session

instead of:

```
at (?myEvent((var x), (var y), 1))
```

urbiscript
Session

- Waituntil and bindings.

Bindings performed in `waituntil` constructs are now available in its context:

```
waituntil(?event(var x));
// x is available
echo (x);
```

urbiscript
Session

- `List.insert`.

Now uses an index as its first argument and inserts the given element before the index position:

```
["a", "b", "c"].insert(1, "foo");
[00000001] ["a", "foo", "b", "c"]
```

urbiscript
Session

- `List.sort`.

Now takes an optional argument, which is a function to call instead of the `<` operator. Here are two examples illustrating how to sort strings, depending on whether we want to be case-sensitive (the default) or not:

```
["foo", "bar", "Baz"].sort;
[00000001] ["Baz", "bar", "foo"]
["foo", "bar", "Baz"].sort(function(x, y) {x.toLowerCase < y.toLowerCase});
[00000002] ["bar", "Baz", "foo"]
```

urbiscript
Session

- `System.searchPath`.

It is now possible to get the search path for files such as ‘urbi.u’ or ‘URBI.INI’ by using `System.searchPath`.

- `System.getenv`.

Now returns `nil` if a variable cannot be found in the environment instead of `void`. This allows you do to things such as:

urbiscript
Session

```
var ne = System.getenv("nonexistent");
if (!ne.isNil) do_something(ne);
```

while previously you had to retrieve the environment variable twice, once to check for its existence and once to get its content.

- **Control.disown.**

It is now possible to start executing code in background while dropping all the tags beforehand, including the connection tag. The code will still continue to execute after the connection that created it has died.

- **Object.removeSlot.**

Now silently accepts non-existing slot names instead of signaling an error.

- **Semaphore.criticalSection.**

It is now possible to define a critical section associated with a semaphore. The **Semaphore.acquire** method will be called at the beginning, and if after that the operation is interrupted by any means the **Semaphore.release** operation will be called before going on. If there are no interruption, the **Semaphore.release** operation will also be called at the end of the callback:

urbiscript
Session

```
var s = \refSlot[Semaphore]{new}(1);
s.criticalSection(function () { echo ("In the critical section") });
```

- **System.stats.**

Its output is now expressed in seconds rather than milliseconds, for consistency with the rest of the kernel.

36.22.3 UObjects

- **void.**

The error message given to the user trying to cast a void UVar has been specialized.

Remote bound methods can now return void.

- **Coroutine interface.**

The functions **yield()**, **yield_until()**, and **yield_until_things_changed()** have been added to the UObject API. They allow the user to write plugin UObject code that behaves like any other coroutine in the kernel: if **yield()** is called regularly, the kernel can continue to work while the user code runs. Meaningful implementation for these functions is provided also in remote mode: calling **yield()** will allow the UObject remote library to process pending messages from within the user callback.

- **Remote UObject initialization.**

Remote UObject instantiation is now atomic: the API now ensures that all variables and functions bound from the UObject constructor and init are visible as soon as the UObject itself is visible. Code like:

urbiscript
Session

```
waituntil(uobjects.hasSlot("MyRemote")) | var m = \refSlot[MyRemote]{new}();
```

is now safe.

36.22.4 Auxiliary programs

urbi-launch Now, options for **urbi-launch** are separated from options to give to the underlying program (in remote and start modes) by using ‘--’. Use ‘**urbi-launch --help**’ to get the full usage information.

36.23 Urbi SDK 2.0 beta 2

Released on 2008-11-03.



36.23.1 urbiscript

2008-11-03.

- `object` and `from` as identifiers.

`object` and `from` are now regular identifiers and can be used as other names. For example, it is now legal to declare:

```
var object = 1;
var from = 1;
```

urbiscript
Session

- Hexadecimal literals.

It is now possible to enter (integral) hexadecimal numbers by prefixing them with `0x`, as in:

```
0x2a;
[00000001] 42
```

urbiscript
Session

Only integral numbers are supported.

36.23.2 Standard library

- `String.asList`.

`String` now has a `asList` method, which can be used transparently to iterate over the characters of a string:

```
for (var c: "foo") echo (c);
[00000001] *** f
[00000002] *** o
[00000003] *** o
```

urbiscript
Session

- `String.split` method Largely improved.

- `List.min` and `List.max`.

It is now possible to call `min` and `max` on a list. By default, the `<` comparison operator is used, but one explicit “less than” function can be provided as `min` or `max` argument should one be needed. Here is an example on how to compare strings in case-sensitive and case-insensitive modes:

```
["the", "brown", "Fox"].min;
[00000001] "Fox"
["the", "brown", "Fox"].min(function (l, r) { l.toLowerCase < r.toLowerCase });
[00000002] "brown"
```

urbiscript
Session

`Math.min` and `Math.max` taking an arbitrary number of arguments have also been defined. In this case, the default `<` operator is used for comparison:

```
min(3, 2, 17);
[00000001] 2
```

urbiscript
Session

- Negative indices.

It is now possible to use negative indices when taking list elements. For example, `-1` designates the last element, and `-2` the one before that.

```
["a", "b", "c"][-1];
[00000001] "c"
```

urbiscript
Session

- Tag names.

Tags were displayed as `Tag_0x01234500` which did not make their `name` slot apparent. They are now displayed as `Tag<name>`:

urbscript
Session

```
Tag.new;
[00000001] Tag<tag_1>
Tag.new("mytag");
[00000002] Tag<mytag>
```

- `every` and exceptions.

If an exception is thrown and not caught during the execution of an `every` block, the `every` expression is stopped and the exception displayed.

36.23.3 UObjects

`UVar::type()` method.

It is now possible to get the type of a `UVar` by calling its `type()` method, which returns a `UDataType` (see ‘`urbi/uvalue.hh`’ for the types declarations).

36.23.4 Run-time

Stack exhaustion check on Windows

As was done on GNU/Linux already, stack exhaustion condition is detected on Windows, for example in the case of an infinite recursion. In this case, `SchedulingError` will be raised and can be caught.

Errors from the trajectory generator are propagated

If the trajectory generator throws an exception, for example because it cannot assign the result of its computation to a non-existent variable, the error is propagated and the generator is stopped:

urbscript
Session

```
xx = 20 ampli:5 sin:10s;
[00002140:error] !!! lookup failed: xx
```

36.23.5 Bug fixes

Support for Windows shares

Previous versions of the kernel could not be launched from a Windows remote directory whose name is starting with two slashes such as ‘`//share/some/dir`’.

Implement `UVar::syncValue()` in plugged uobjects

Calling `syncValue()` on a `UVar` from a plugged `UObject` resulted in a link error. This method is now implemented, but does nothing as there is nothing to do. However, its presence is required to be able to use the same `UObject` in both remote and engine modes.

`isdef` works again

The support for k1 compatibility function `isdef` was broken in the case of composed names or variables whose content was `void`. Note that we do not recommend using `isdef` at all. Slots related methods such as `getSlot`, `hasSlot`, `locateSlot`, or `slotNames` have much cleaner semantics.

`__name` macro

In some cases, the `__name` macro could not be used with plugged uobjects, for example in the following expression:

urbscript
Session

```
send(__name + ".val = 1;");
```

This has been fixed. `__name` contains a valid slot name of `uobjects`.

36.23.6 Auxiliary programs

The sample programs demonstrating the SDK Remote, i.e., how to write a client for the Urbi server, have been renamed from `urbi*` to `urbi-*`. For instance `urbisend` is now spelled `urbi-send`.

Besides, their interfaces are being overhauled to be more consistent with the Urbi command-line tool-box. For instance while `urbisend` used to require exactly two arguments (host-name, file to send), it now supports options (e.g., ‘`--help`’, ‘`--port`’ to specify the port etc.), and as many files as provided on the command line.

Chapter 37

Licenses

Part of the Urbi SDK is based on software distributed under the following licenses. Other licenses are included below for information; each section explains why its corresponding license is included here.

37.1 Boost Software License 1.0

Urbi SDK uses Boost libraries. Boost libraries and headers are included in binary packages. See <http://www.boost.org/users/license.html>.

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

37.2 BSD License

In order to be included in Urbi SDK, contributors can use this license for their patches, see [Section 18.5.2](#).

This license is also known as:

- the “modified BSD License,” see the [License List](#) maintained by the Free Software Foundation;
- or the “BSD three-clause License,” see <http://www.opensource.org/licenses/BSD-3-Clause>.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

37.3 Expat License

In order to be included in Urbi SDK, contributors can use this license for their patches, see [Section 18.5.2](#).

This license is also known as the “MIT License,” see the [License List](#) maintained by the Free Software Foundation.

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

37.4 gnu.bytecode

The Java API for UObject uses the [gnu.bytecode](#) Java package.

The software (with related files and documentation) in these packages are copyright (C) 1996-2006 Per Bothner.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

37.5 ICU License

Urbi SDK uses Boost libraries which in turn use ICU. Boost and ICU are included in binary packages. See <http://icu-project.org/repos/icu/icu/trunk/license.html>.

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2011 International Business Machines Corporation and others

All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

37.6 Independent JPEG Group's Software License

Urbi SDK uses the Independent JPEG Group's software, better known as libjpeg. Its license requires us to state that:

This software is based in part on the work of the Independent JPEG Group.

The license below is the relevant part of the 'COPYRIGHT' file in the top-level directory of libjpeg.

In plain English:

1. We don't promise that this software works. (But if you find any bugs, please let us know!)
2. You can use this software for whatever you want. You don't have to pay us.
3. You may not pretend that you wrote this software. If you use it in a program, you must acknowledge somewhere in your documentation that you've used the IJG code.

In legalese:

The authors make NO WARRANTY or representation, either express or implied, with respect to this software, its quality, accuracy, merchantability, or fitness for a particular purpose. This software is provided "AS IS", and you, its user, assume the entire risk as to its quality and accuracy.

This software is copyright (C) 1991-2012, Thomas G. Lane, Guido Vollbeding.
All Rights Reserved except as specified below.

Permission is hereby granted to use, copy, modify, and distribute this software (or portions thereof) for any purpose, without fee, subject to these conditions:

- (1) If any part of the source code for this software is distributed, then this README file must be included, with this copyright and no-warranty notice unaltered; and any additions, deletions, or changes to the original files must be clearly indicated in accompanying documentation.
- (2) If only executable code is distributed, then the accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group".
- (3) Permission for use of this software is granted only if the user accepts full responsibility for any undesirable consequences; the authors accept NO LIABILITY for damages of any kind.

These conditions apply to any software derived from or based on the IJG code, not just to the unmodified library. If you use our work, you ought to acknowledge us.

Permission is NOT granted for the use of any IJG author's name or company name in advertising or publicity relating to this software or products derived from it. This software may be referred to only as "the Independent JPEG Group's software".

We specifically permit and encourage the use of this software as the basis of commercial products, provided that all warranty or liability claims are assumed by the product vendor.

37.7 Libcoroutine License

Urbi uses the libcoroutine from the *Io* language. See <http://www.iolangue.com/>.

```
(This is a BSD License)

Copyright (c) 2002, 2003 Steve Dekorte
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the author nor the names of other contributors may be
  used to endorse or promote products derived from this software without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 DAMAGE.
```

37.8 OpenSSL License

LICENSE ISSUES

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

```
/* ===== */
* Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
*
* 1. Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in
*    the documentation and/or other materials provided with the
*    distribution.
```

```

*
* 3. All advertising materials mentioning features or use of this
*    software must display the following acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For written permission, please contact
*    openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*    nor may "OpenSSL" appear in their names without prior written
*    permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*    acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to. The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code. The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without

```

```

* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*      "This product includes cryptographic software written by
*      Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the routines from the library
*    being used are not cryptographic related :).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*      "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.

*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

37.9 ROS

The ROS bridge ([Chapter 16](#)) using ROS libraries, which are included in our binary packages. As of today (October 10th, 2011), the ROS guidelines for developers (<http://www.ros.org/wiki/DevelopersGuide#Licensing>) point to a BSD two-clause license (<http://www.opensource.org/licenses/bsd-license.php>). This is the same license as the three-clause BSD ([Section 37.2](#)), without the third-clause.

```

Copyright (c) 2010, Willow Garage
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in
  the documentation and/or other materials provided with the
  distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT

```

HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

37.10 Urbi Open Source Contributor Agreement

The following text is not a license, but it's one way for Urbi SDK contributors to enable us to use their code. See [Section 18.5.2](#). This document itself is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License <http://creativecommons.org/licenses/by-sa/3.0/>.

IMPORTANT - PLEASE READ CAREFULLY BEFORE SIGNING

These terms apply to your contribution of materials to the Urbi Open Source project owned and managed by us (Gostai S.A.S), and set out the intellectual property rights you grant to us in the contributed materials. If this contribution is on behalf of a company, the term 'you' will also mean the company you identify below. If you agree to be bound by these terms, fill in the information requested below and provide your signature.

1. The term "contribution" means any source code, object code, patch, tool, creation, images, sound, sample, graphic, specification, manual, documentation, or any other material posted or submitted by you to the Urbi Open Source project, in human or machine readable form. For the avoidance of doubt: is considered as a "contribution" any material that you will send to us via one of the email addresses of the Gostai employees or owned projects (for example *project-contrib@gostai.com* as displayed on the project web page), or via pushing to any of our public git/svn or other code repositories.
 - you hereby assign to us joint ownership, and to the extent that such assignment is or becomes invalid, ineffective or unenforceable, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free, unrestricted license to exercise all rights under those copyrights, including a license to use, reproduce, prepare derivative works of, publicly display, publicly perform and distribute. This also includes, at our option, the right to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements;
 - you agree that each of us can do all things in relation to your contribution as if each of us were the sole owners, and if one of us makes a derivative work of your contribution, the one who makes the derivative work (or has it made) will be the sole owner of that derivative work;
 - you agree that you will not assert any moral rights in your contribution against us, our licensees or transferees;
 - you agree that we may register a copyright in your contribution and exercise all ownership rights associated with it; and
 - you agree that neither of us has any duty to consult with, obtain the consent of, pay or render an accounting to the other for any use or distribution of your contribution.
2. With respect to any patents you own, or that you can license without payment to any third party, you hereby grant to us a perpetual, irrevocable, non-exclusive, worldwide, no-charge, royalty-free license to:
 - make, have made, use, sell, offer to sell, import, and otherwise transfer your contribution in whole or in part, alone or in combination with or included in any product, work or materials arising out of the project to which your contribution was submitted, and
 - at our option, to sublicense these same rights to third parties through multiple levels of sublicensees or other licensing arrangements.

3. Except as set out above, you keep all right, title, and interest in your contribution. The rights that you grant to us under these terms are effective on the date you first submitted a contribution to us, even if your submission took place before the date you sign these terms.
4. With respect to your contribution, you represent that:
 - it is an original work and that you can legally grant the rights set out in these terms;
 - it does not to the best of your knowledge violate any third party's copyrights, trademarks, patents, or other intellectual property rights; and
 - you are authorized to sign this contract on behalf of your company (if identified below).
5. These terms will be governed in all respects by French laws and the French courts only shall have jurisdiction in relation to them.

If available, please list your user name(s) (or "login") and the name of the project(s) (or project website(s) or repository) for which you would like to contribute materials.

Your user name:

Project name, website or repository:

Your user name:

Project name, website or repository:

Your contact information (Please print clearly):

Your name:

Your company's name (if applicable):

Mailing address:

Telephone, Fax and Email:

Your signature:

Date:

To deliver these terms to us, scan and email, or fax a signed copy to us using the contrib@gostai.com email address or fax number set out on the appropriate project website.

Chapter 38

Bibliography

- [1] Jean-Christophe Baillie. URBI: A universal language for robotic control. *International journal of Humanoid Robotics*, October 2004.
- [2] Jean-Christophe Baillie. URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825, 2005.
- [3] Jean-Christophe Baillie. Design principles for a universal robotic software platform and application to URBI. In Davide Brugali, Christian Schlegel, Issa A. Nesnas, William D. Smart, and Alexander Braendle, editors, *IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR-II)*, SDIR-II, Roma, Italy, April 2007. IEEE Robotics and Automation Society.
- [4] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Events! (reactivity in urbiscript). In Serge Stinckwich, editor, *First International Workshop on Domain-Specific Languages and models for ROBotic systems*, October 2010.
- [5] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Tag: Job control in urbiscript. In Noury Bouraqadi, editor, *Fifth National Conference on Control Architecture of Robots*, May 2010.
- [6] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, Matthieu Nottale, and Samuel Tardieu. The Urbi universal platform for robotics. In Itsuki Noda, editor, *First International Workshop on Standards and Common Platform for Robotics*, November 2008.

Chapter 39

Glossary

This chapter aggregates the definitions used in the this document.

Bioloid The Robotis Bioloid is a hobbyist and educational robot kit produced by the Korean robot manufacturer Robotis. The Bioloid platform consists of components and small, modular servomechanisms called Dynamixels, which can be used in a daisy-chained fashion to construct robots of various configurations, such as wheeled, legged, or humanoid robots. The Bioloid system is thus comparable to the LEGO Mindstorms and VEXplorer kits.

Gostai Console This tool provides a graphical user interface to a remote Urbi server (see [Figure 39.1](#)). Unix users (GNU/Linux or Mac OS X) can use the traditional `telnet` tool. Windows users are invited to use Gostai Console instead. See [Chapter 3](#).

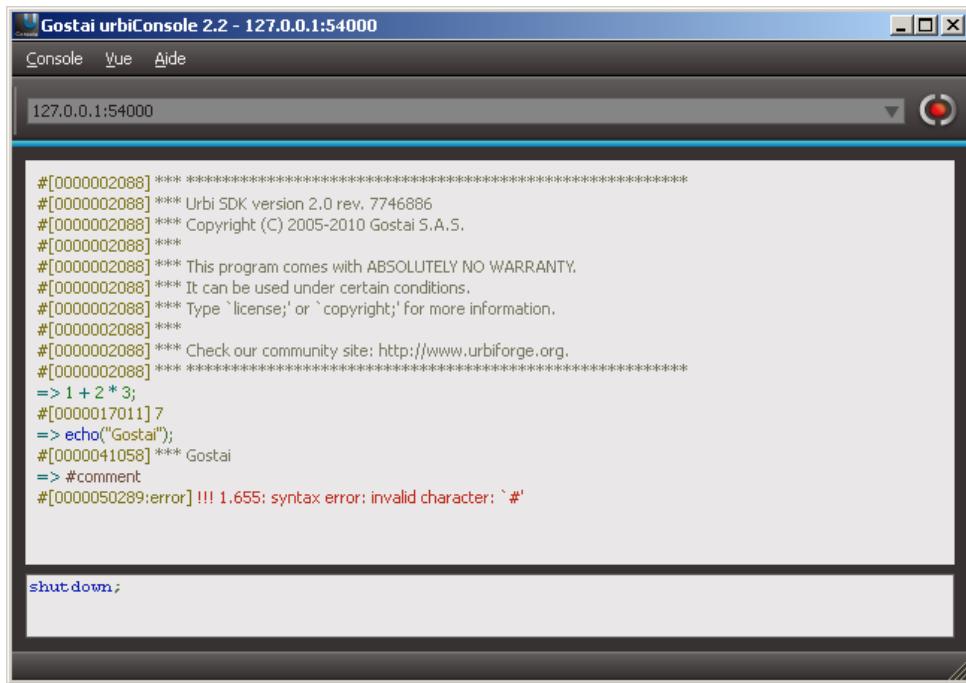


Figure 39.1: Gostai Console

Gostai Editor Also called UEdit, Gostai urbiscript Editor ([Figure 39.2](#)) is a lightweight urbiscript source code editor with semantic highlighting. It is available for Windows and GNU/Linux platforms.

Gostai Lab This tool (see [Figure 39.3](#)), which includes the features of Gostai Console, allows to build easily elaborate remote controller for robots. It provides various widgets to visualize data from the robot (including video and sound), and to modify the state of the robot.

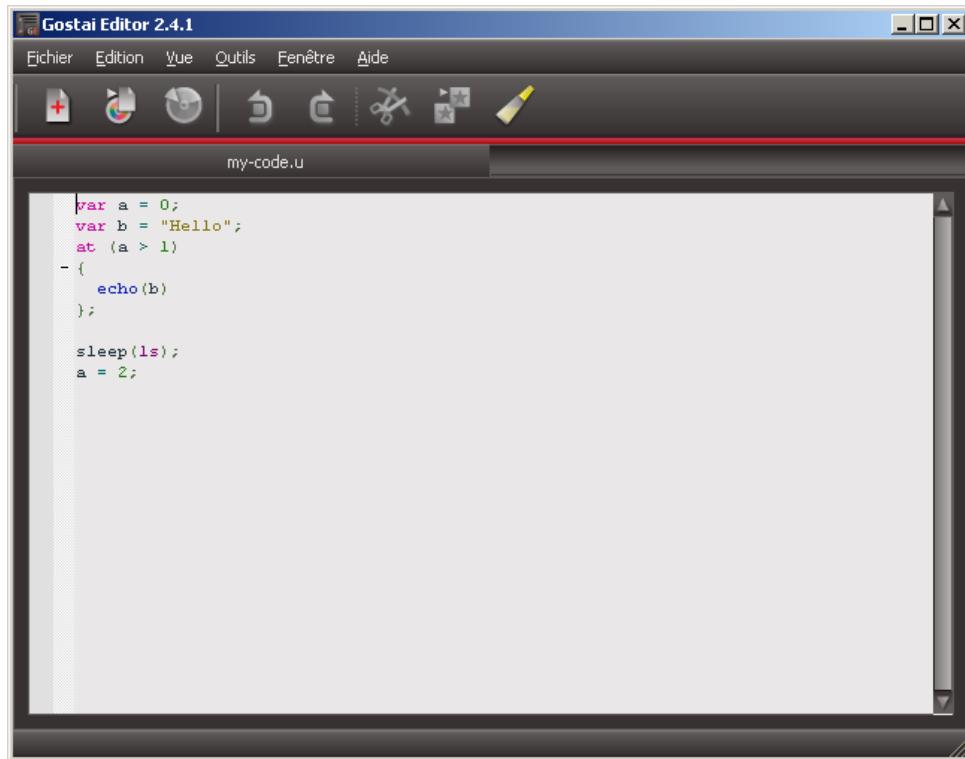


Figure 39.2: Gostai Editor

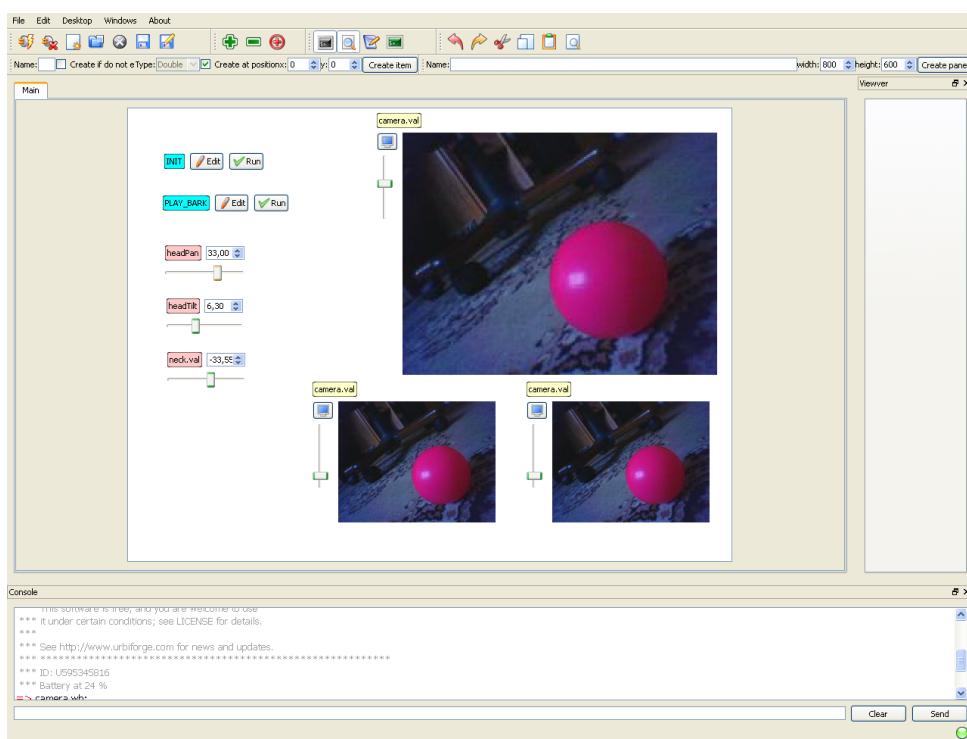


Figure 39.3: Gostai Lab

Gostai Studio This tool (see [Figure 39.4](#)), includes all the features of Gostai Console and Gostai Lab. It is a high-level Integrated Development Environment for Urbi. Its formalism is based on *Hierarchical Finite State Machines*.

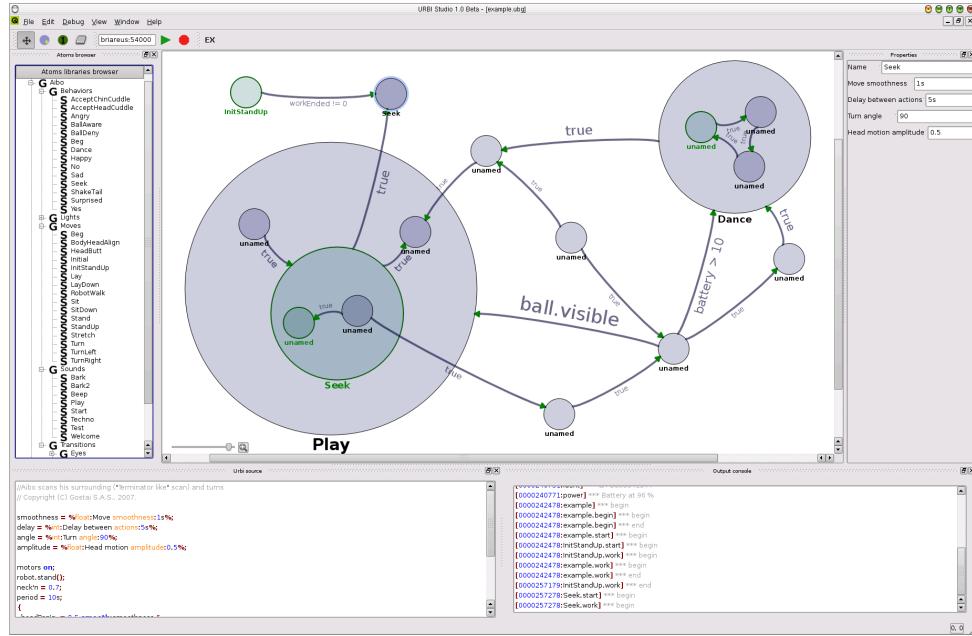


Figure 39.4: Gostai Studio

RMP The Segway Robotic Mobility Platform is a robotic platform based on the Segway Personal Transporter. See <http://rmp.segway.com>.

RTP Real-time Transport Protocol (RFC 3550). A protocol designed for streaming (for instance audio/video).

ROS Robot Operating System, <http://www.ros.org/>, developed by Willow Garage, <http://www.willowgarage.com/>. It is an abstraction layer on top of the genuine operating system (such as GNU/Linux) that provides hardware abstraction, device control, common algorithms, message-passing between processes, and package management.

Spykee The Spykee is a WiFi-enabled robot built by Meccano (known as Erector in the United States). It is equipped with a camera, speaker, microphone, and moves using two tracks. See <http://www.spykeeworld.com>.

urbi-console Former name of “Gostai Console”. See that item.

Chapter 40

List of Tables

23.1 Keywords	204
23.2 Angle units	204
23.3 Duration units	205
23.4 String escapes	206
23.5 Arithmetic operators	211
23.6 Assignment operators	211
23.7 Postfix operators	213
23.8 Bitwise operators	214
23.9 Boolean operators	214
23.10 Comparison operators	216
23.11 Container operators	216
23.12 Object operators	217
23.13 Operators summary	218

Chapter 41

List of Figures

1.1 A Bird-View of the Urbi Architecture	9
16.1 Output from <code>rxgraph</code>	143
30.1 A Gostai Console session.	482
30.2 Jazz Internal Website.	482
30.3 User services web page.	483
30.4 User services web page: <i>test.u</i> service was uploaded.	483
30.5 User services web page: <i>test.u</i> service loaded at startup.	484
30.6 System page: display Jazz software information and allow updates.	484
30.7 Service activation page: tune Jazz software profile.	485
33.1 Installing the Spykee interface	513
39.1 Gostai Console	579
39.2 Gostai Editor	580
39.3 Gostai Lab	580
39.4 Gostai Studio	581

Chapter 42

Index

Symbols

'[287](#)
'\$id'
 Object.[~](#), [355](#)
'*'
 Float.[~](#), [305](#)
 List.[~](#), [335](#)
 String.[~](#), [393](#)
 Tuple.[~](#), [419](#)
'**'
 Float.[~](#), [305](#)
'*='
 Object.[~](#), [360](#)
'+'
 Binary.[~](#), [258](#)
 Date.[~](#), [271](#)
 Float.[~](#), [305](#)
 List.[~](#), [336](#)
 Position.[~](#), [368](#)
 String.[~](#), [393](#)
 Tuple.[~](#), [419](#)
'+='
 List.[~](#), [336](#)
 Object.[~](#), [360](#)
'-'
 Date.[~](#), [271](#)
 Float.[~](#), [304](#)
 List.[~](#), [336](#)
 Position.[~](#), [368](#)
'-='
 Object.[~](#), [360](#)
'/'
 Directory.[~](#), [279](#)
 Float.[~](#), [305](#)
 Path.[~](#), [365](#)
'/= '
 Object.[~](#), [360](#)
'<'
 Date.[~](#), [271](#)
 Float.[~](#), [304](#)
List.[~](#), [337](#)
Path.[~](#), [366](#)
Position.[~](#), [368](#)
String.[~](#), [393](#)
Tuple.[~](#), [418](#)
'<<'
 Channel.[~](#), [264](#)
 Directory.[~](#), [279](#)
 Event.[~](#), [288](#)
 Float.[~](#), [304](#)
 Group.[~](#), [317](#)
 List.[~](#), [336](#)
 Logger.[~](#), [346](#)
 Ros.Topic.[~](#), [425](#)
'=='
 Binary.[~](#), [258](#)
 Date.[~](#), [271](#)
 Dictionary.[~](#), [275](#)
 Float.[~](#), [305](#)
 List.[~](#), [335](#)
 Location.[~](#), [344](#)
 Object.[~](#), [360](#)
 Path.[~](#), [366](#)
 Position.[~](#), [368](#)
 String.[~](#), [393](#)
 Tuple.[~](#), [419](#)
'==='
 Object.[~](#), [361](#)
'>>'
 Float.[~](#), [304](#)
'[]'
 Dictionary.[~](#), [275](#)
 List.[~](#), [335](#)
 Regexp.[~](#), [381](#)
 String.[~](#), [394](#)
 Tuple.[~](#), [418](#)
'[]='
 Dictionary.[~](#), [275](#)
 List.[~](#), [335](#)
 String.[~](#), [394](#)

Tuple. \sim , 418
 $,$ \sim ,
Float. \sim , 304
 $,$ $=$,
Object. \sim , 360
'assert'
System. \sim , 395
'bitand'
Float. \sim , 299
'bitor'
Float. \sim , 299
'each&'
Float. \sim , 300
List. \sim , 329
RangeIterable. \sim , 379
'emit'
Event. \sim , 287
'new'
Singleton. \sim , 384
 $::=$, 532
_exit
System. \sim , 394

A

aborted
Interface.Mobile. \sim , 437
aborted?
jazz. \sim , 494
abs
Float. \sim , 298
Math. \sim , 348
absolute
Path. \sim , 364
Accel, 412
TrajectoryGenerator. \sim , 416
accel, 445
acceleration
Interface.AccelerationSensor. \sim , 438
AccelerationSensor, 438
acceptVoid
Object. \sim , 351
void. \sim , 422
accumulate, 329
acos
Float. \sim , 298
Math. \sim , 348
acquire
Semaphore. \sim , 382
add
Group. \sim , 316
addComponent
Component. \sim , 447
addDevice

Component. \sim , 447
addProto
Object. \sim , 351
advertise
Ros.Topic. \sim , 426
aec
jazz.body.head. \sim , 489
agc
jazz.body.head. \sim , 489
age
Interface.TextToSpeech. \sim , 440
aggressive_mode
uobjects.EchoCanceller. \sim , 499
Aldebaran, 453
aliasing, 120
alignment
FormatInfo. \sim , 307
aliveJobs
System. \sim , 394
all
RangeIterable. \sim , 378
'all.stamp', 183
allProto
Object. \sim , 351
allSlotNames
Object. \sim , 352
alsa
jazz.body.head. \sim , 489
alt
FormatInfo. \sim , 307
angle, 203
angle
Interface.RotationalMotor. \sim , 438
angleMax
Interface.Laser. \sim , 439
jazz.body.laser. \sim , 497
P3dx.body.laser. \sim , 508
angleMin
Interface.Laser. \sim , 439
jazz.body.laser. \sim , 497
P3dx.body.laser. \sim , 508
ankle, 443
any
RangeIterable. \sim , 379
append
List. \sim , 327
apply
Code. \sim , 267
Object. \sim , 352
Primitive. \sim , 369
argMax
List. \sim , 327
argMin

List. \sim , 328
 args
 CallMessage. \sim , 261
 argsCount
 CallMessage. \sim , 261
 Argument
 Exception. \sim , 289
 arguments
 System. \sim , 394
 ArgumentType
 Exception. \sim , 290
 Arity
 Exception. \sim , 290
 arity, 135
 arm, 443
 as
 Object. \sim , 352
 asBool
 Boolean. \sim , 260
 Dictionary. \sim , 275
 Float. \sim , 298
 List. \sim , 328
 Object. \sim , 352
 ASCII, 201
 asEvent
 Event. \sim , 287
 asExecutable
 Executable. \sim , 292
 asFloat
 Date. \sim , 271
 Duration. \sim , 284
 Float. \sim , 298
 Hash. \sim , 317
 String. \sim , 389
 asin
 Float. \sim , 298
 Math. \sim , 348
 asJob
 Job. \sim , 321
 asList
 Dictionary. \sim , 275
 Directory. \sim , 279
 Enumeration. \sim , 285
 File. \sim , 293
 Float. \sim , 298
 Formatter. \sim , 309
 List. \sim , 328
 Path. \sim , 364
 String. \sim , 389
 asMutex
 Mutex. \sim , 350
 asPath
 Directory. \sim , 280
 File. \sim , 293
 asPrimitive
 Primitive. \sim , 369
 asPrintable
 File. \sim , 293
 Path. \sim , 364
 Regexp. \sim , 380
 String. \sim , 390
 asProcess
 Process. \sim , 370
 assert_
 System. \sim , 395
 assert_op
 System. \sim , 395
 assertion, 242
 assertion frame, 533
 assignment, 211
 associative array, 273
 asString
 Binary. \sim , 259
 Code. \sim , 267
 Date. \sim , 271
 Dictionary. \sim , 276
 Directory. \sim , 280
 Duration. \sim , 284
 Enumeration. \sim , 286
 File. \sim , 293
 Float. \sim , 299
 Group. \sim , 316
 Job. \sim , 321
 Lazy. \sim , 326
 List. \sim , 328
 Location. \sim , 344
 Path. \sim , 364
 Position. \sim , 368
 Process. \sim , 371
 Regexp. \sim , 380
 StackFrame. \sim , 388
 String. \sim , 390
 Tuple. \sim , 418
 asuobjects
 uobjects. \sim , 419
 asynchronous
 UConnection. \sim , 60
 at, 243
 atan
 Float. \sim , 299
 Math. \sim , 348
 atan2
 Math. \sim , 348
 attribute, 254
 attributes, 44
 AudioIn, 433

A
 AudioOut, 434
 authors
 Lobby.~, 339
 AutomaticGainControl, 499
 AX12, 464
 AXS1, 465

B
 b
 Interface.RGBLed.~, 435
 jazz.body.head.eye.~, 488
 back
 List.~, 328
 backtrace
 Exception.~, 288
 Job.~, 321
 System.~, 395
 Traceable.~, 411
 BadInteger
 Exception.~, 290
 BadNumber
 Exception.~, 290
 banner
 Lobby.~, 339
 Barrier, 257
 Global.~, 310
 basename
 Directory.~, 282
 File.~, 293
 Path.~, 364
 Battery, 434, 476
 Beeper, 477
 begin
 Location.~, 345
 Tag.~, 409
 Binary, 258
 Global.~, 310
 Binding
 Pattern.~, 366
 binding, 44
 bindings
 Pattern.~, 366
 Bioloid, 463
 blob, 258, 434
 BlobDetector, 434
 block, 405
 catch-block, 238
 try-block, 238
 block
 Tag.~, 409
 blocked
 Tag.~, 409
 blocking, 49

body, 442
 P3dx.~, 506
 bodyString
 Code.~, 267
 Boolean, 259
 bootstrap, 175
 bounce
 Object.~, 352
 buffer, 362
 bufferSize
 uobjects.AlsaSpeaker.~, 498
 build, 175
 build directory, 180
 builddir, 180
 buzzerIndex
 AXS1.~, 465
 buzzerTime
 AXS1.~, 465
 bytesReceived
 Lobby.~, 340
 bytesSent
 Lobby.~, 340

C
 ‘-C’, 196, 200
 ‘-c’, 195–197, 199
 caching, 325
 calibrating
 uobjects.EchoCanceller.~, 499
 callCount
 UConnection.~, 60
 caller, 263
 CallMessage, 260
 Global.~, 310
 callMessage
 Object.~, 352
 calls
 Profile.Function.~, 376
 Profile.~, 375
 camera, 445
 capacity
 Interface.Battery.~, 434
 cd
 Path.~, 364
 ceil
 Float.~, 299
 cerr
 Global.~, 310
 Channel, 263
 Global.~, 310
 ‘--check’, 196
 checkMaster
 Ros.~, 423

clapCount
 AXS1.~, 465
class, 351
 '--classpath=*path*', 196
clavicle, 445
 '--clean', 199
clear
 Dictionary.~, 276
 Directory.~, 280
 List.~, 328
 uobjects.AlsaSpeaker.~, 498
clearStats
 uobjects.~, 419
 'CLIENT.INI', 191
clog
 Global.~, 310
clone
 Float.~, 299
 Job.~, 321
 Object.~, 352
 Singleton.~, 384
cloneSlot
 Object.~, 352
close
 Stream.~, 388
closeDelay
 uobjects.AlsaSpeaker.~, 498
ClosedLoop
 TrajectoryGenerator.~, 416
closest
 String.~, 390
closure, 114
closure (lexical), 222
Code, 265
 Global.~, 310
code
 CallMessage.~, 261
column
 Position.~, 368
columns
 Position.~, 368
combine
 Hash.~, 318
Command, 477
commands
 Kernel1.~, 323
comment, 95, 201, 531
Comparable, 267
 Global.~, 310
compl
 Float.~, 299
Component, 447
components, 429
connect
 Socket.~, 386
connected
 Lobby.~, 340
 Socket.~, 386
connection
 Server.~, 383
connections
 Kernel1.~, 323
connectionStats
 uobjects.~, 420
connectionTag
 Lobby.~, 340
connectSerial
 Socket.~, 386
Constness
 Exception.~, 290
constructor, 163
Container, 268
content
 Directory.~, 280
 File.~, 293
Control, 269
Control.detach, 128
copy
 Directory.~, 280
 File.~, 293
 Kernel1.~, 323
 UVar.~, 420
copy on write, 227
copy-on-write, 118
copyInto
 Directory.~, 280
 File.~, 293
copyright
 Lobby.~, 341
copyrightHolder
 System.PackageInfo.~, 403
copyrightYears
 System.PackageInfo.~, 403
copySlot
 Object.~, 353
 '--core=*core*', 200
Cos, 413
cos
 Float.~, 299
 Math.~, 348
 '--count=*count*', 197
cout
 Global.~, 310
coveredAngle
 P3dx.body.odometry.~, 506
coveredDistance

P3dx.body.odometry.~, 506
create
 Directory.~, 281
 File.~, 294
 Lobby.~, 341
createAll
 Directory.~, 281
createSlot
 Object.~, 353
criticalSection
 Semaphore.~, 382
current
 Interface.Battery.~, 434
 '--customize=*file*', 195
cwd
 Path.~, 364
cwLimit, ccwLimit
 AX12.~, 464
cycle
 System.~, 395

D

'-D', 198
 '-d', 191, 194–196, 198
 '-Dsymbol', 200
data
 Binary.~, 259
Date, 270
 Global.~, 310
day
 Date.~, 272
debug
 Logger.~, 346
 '--debug', 199
 '--debug=*level*', 191, 195, 196
decrement
 Interface.AutomaticGainControl.~, 499
 '--deep-clean', 199
delay
 uobjects.EchoCanceller.~, 499
delete, 158
denoise
 jazz.body.head.~, 489
Denoiser, 500
destructor, 295
detach
 Control.~, 269
 Global.~, 310
 '--device=*device*', 194, 198
devices, 430
devices
 Kernel1.~, 323
DGain

Interface.Motor.~, 437
 dictionaries, 203
 Dictionary, 273
 Global.~, 311
 dictionary, 273
digits
 Float.limits.~, 305
digits10
 Float.limits.~, 305
Directory, 278, 279, 365
Directory, 278
 Global.~, 311
dirname
 Path.~, 365
 '--disable-automain', 200
 '--disable-bindings', 180
disconnect
 Socket.~, 386
 UConnection.~, 60
disconnected
 Socket.~, 386
disown
 Control.~, 269
 Global.~, 311
distance
 Interface.DistanceSensor.~, 438
 String.~, 390
distanceMax
 Interface.Laser.~, 439
 jazz.body.laser.~, 497
distanceMin
 Interface.Laser.~, 439
 jazz.body.laser.~, 497
DistanceSensor, 438
dnsAddress
 jazz.network.~, 490
 'doc/tests/test-suite.log', 184
done
 Process.~, 371
dump
 Component.~, 447
 Logger.~, 346
 Object.~, 353
dumpState
 Job.~, 321
Duration, 284
 Global.~, 311
duration, 203
duration
 Interface.AudioIn.~, 433
 '--duration=*duration*', 198

‘-e’, 192, 197
 each
 Float.~, 299
 Group.~, 316
 List.~, 328
 RangeIterable.~, 379
 each&
 Group.~, 316
 eachi
 List.~, 329
 ear, 444
 EBNF, 532
 echo, 96
 Channel.~, 264
 Global.~, 311
 Lobby.~, 341
 echoEach
 Lobby.~, 341
 elbow, 443
 elementAdded
 Dictionary.~, 276
 elementChanged
 Dictionary.~, 276
 elementRemoved
 Dictionary.~, 276
 elongation
 Interface.BlobDetector.~, 434
 empty
 Binary.~, 259
 Dictionary.~, 276
 Directory.~, 281
 List.~, 329
 String.~, 390
 ‘--enable-compilation-mode=*mode*’, 180
 ‘--enable-doc-sections=*sections*’, 180
 ‘--enable-doc=*formats*’, 180
 ‘--enable-examples’, 180
 ‘--enable-library-suffix=*suffix*’, 180
 ‘--enable-sdk-remote’, 180
 ‘--enable-ufloat=*kind*’, 180
 enabled
 Channel.~, 264
 UConnection.~, 60
 enableStats
 uobjects.~, 420
 encoding, 201
 end
 Location.~, 345
 Tag.~, 409
 energy_peak
 uobjects.EchoCanceller.~, 499
 engine, 3
 enter
 Tag.~, 409
 Enumeration, 285
 environment variable, 531
 epoch
 Date.~, 272
 epsilon
 Float.limits.~, 306
 erase
 Dictionary.~, 276
 err
 Logger.~, 346
 error
 Socket.~, 386
 essid
 jazz.network.~, 490
 eval
 CallMessage.~, 262
 Lazy.~, 326
 System.~, 396
 evalArgAt
 CallMessage.~, 262
 evalArgs
 CallMessage.~, 262
 evaluate
 Global.~, 311
 evaluation mode, 518
 Event, 287
 Global.~, 311
 event, 165, 287
 events
 Kernel1.~, 323
 every, 245
 Exception, 288
 Global.~, 311
 exception
 catching, 238
 throwing, 237
 Executable, 292
 Global.~, 311
 ‘executables.stamp’, 183
 exists
 Directory.~, 281
 Path.~, 365
 exp
 Float.~, 300
 Math.~, 348
 exposure
 Interface.VideoIn.~, 441
 ‘--expression=*exp*’, 192
 ‘--expression=*script*’, 197
 Extended Backus-Naur Form, 532
 external
 Global.~, 311

EXTRA_CPPFLAGS, 199
 EXTRA_CXXFLAGS, 199
 EXTRA_LDFLAGS, 199
 eye, 445
 eyebrow, 445

F

‘-F’, 191, 194
 ‘-f’, 192, 197
 fallback
 Group.~, 316
 false, 259
 Global.~, 311
 ‘--fast’, 191
 File, 365
 File, 292
 Global.~, 312
 file
 Position.~, 369
 file name, 531
 ‘--file=file’, 192, 197
 fileCreated
 Directory.~, 281
 fileDeleted
 Directory.~, 282
 FileNotFoundException
 Exception.~, 290
 Filter
 Channel.~, 264
 filter
 List.~, 329
 Finalizable, 295
 Global.~, 312
 finalize
 Finalizable.~, 297
 finger, 443
 finished
 Interface.Mobile.~, 437
 finished?
 jazz.~, 494
 fireRate
 UConnection.~, 60
 first
 Pair.~, 363
 Triplet.~, 417
 flatDump
 Component.~, 447
 Float, 297
 Global.~, 312
 Float.limits, 305
 floor
 Float.~, 300
 flush, 362

flush
 OutputStream.~, 362
 foldl
 List.~, 329
 foot, 443
 force
 Interface.LinearMotor.~, 437
 format
 Float.~, 301
 Interface.VideoIn.~, 441
 format info, 306
 ‘--format=format’, 194
 FormatInfo, 306
 Global.~, 312
 Formatter, 308
 Global.~, 312
 formatter, 308, 309
 forward
 RMP.~, 511
 forwardCoeff
 RMP.~, 512
 forwardSpeed
 RMP.~, 511
 forwardSpeedCoeff
 RMP.~, 512
 freeze, 405
 freeze
 Tag.~, 409
 fresh
 Float.~, 301
 String.~, 390
 fromAscii
 String.~, 390
 front
 List.~, 330
 frozen
 Tag.~, 409
 Function
 Profile.~, 375
 function, 219
 lazy, 222
 return value, 221
 strict, 222
 functions
 Kernel1.~, 323

G

g
 Interface.RGBLed.~, 435
 jazz.body.head.eye.~, 488
 gain
 Interface.AudioIn.~, 433
 Interface.AutomaticGainControl.~, 500

Interface.VideoIn.~, 441
 gainPercent
 jazz.body.head.~, 489
 garbage collection, 171
 gatewayAddress
 jazz.network.~, 491
 GD_CATEGORY, 190
 GD_COLOR, 190
 GD_DISABLE_CATEGORY, 190
 GD_ENABLE_CATEGORY, 190
 GD_LEVEL, 190
 GD_LOC, 190
 GD_NO_COLOR, 190
 GD_PID, 190
 GD_THREAD, 190
 GD_TIME, 190
 GD_TIMESTAMP_US, 190
 gender
 Interface.TextToSpeech.~, 440
 get
 Dictionary.~, 277
 InputStream.~, 318
 getAll
 PubSub.Subscriber.~, 378
 UConnection.~, 60
 getChar
 InputStream.~, 319
 getenv
 System.~, 397
 getIoService
 Server.~, 383
 Socket.~, 386
 getLine
 InputStream.~, 319
 getLocalSlot
 Object.~, 354
 getOne
 PubSub.Subscriber.~, 378
 getPeriod
 Object.~, 353
 getProperty
 Global.~, 312
 Group.~, 316
 Object.~, 353
 getSlot
 Object.~, 354
 getStats
 uobjects.~, 420
 getter function, 58
 getWithDefault
 Dictionary.~, 277
 Global, 309
 Global, 309
 Global.~, 312
 'global.u', 191
 go
 Interface.Mobile.~, 436
 jazz.~, 494
 P3dx.~, 506
 goTo
 Interface.Mobile.~, 436
 jazz.~, 494
 goToAbsolute
 Interface.Mobile.~, 436
 jazz.~, 494
 goToChargingDock
 jazz.~, 494
 goToChargingStation
 Interface.Mobile.~, 436
 grammar fragments, 532
 grip, 443
 Group, 315
 Global.~, 312
 group
 FormatInfo.~, 307
 guard, 231
 gyro, 445
 GyroSensor, 439

H

'-H', 192, 194, 197, 198, 200
 '-h', 191, 194–199
 hand, 443
 has
 Container.~, 268
 Dictionary.~, 277
 Enumeration.~, 286
 List.~, 330
 Regexp.~, 380
 Hash, 317
 hash, 273, 317
 hash
 Float.~, 301
 List.~, 330
 Object.~, 354
 String.~, 391
 hasLocalSlot
 Object.~, 355
 hasNot
 Container.~, 268
 hasProperty
 Group.~, 317
 Object.~, 355
 hasSame
 List.~, 330
 hasSlot

Group.~, 317
Object.~, 355
head, 444
 List.~, 330
hear
 Interface.SpeechRecognizer.~, 440
height
 Interface.VideoIn.~, 441
 jazz.camera.~, 492
‘**--help**’, 191, 194–199
hex
 Float.~, 301
hideSystemFiles
 Traceable.~, 412
Hierarchical Finite State Machines, 581
hip, 443
hookChanged
 UVar.~, 420
host
 Server.~, 383
 Socket.~, 387
 System.Platform.~, 403
‘**--host=address**’, 192
‘**--host=host**’, 194, 197, 198, 200
hostAlias
 System.Platform.~, 403
hostCpu
 System.Platform.~, 403
hostOs
 System.Platform.~, 403
hostVendor
 System.Platform.~, 403
hour
 Date.~, 272

I

‘**-i**’, 192, 197
‘**-Ipath**’, 200
identifier, 202
Identity, 435
IGain
 Interface.Motor.~, 437
immutable, 212
ImplicitTagComponent
 Exception.~, 290
in, 216
increment
 Interface.AutomaticGainControl.~, 499
inf
 Float.~, 302
 Math.~, 348
init
 Dictionary.~, 278

Logger.~, 346
nxt.Battery.~, 476
nxt.Beeper.~, 477
nxt.LightSensor.~, 476
nxt.Servo.~, 475
nxt.SoundSensor.~, 475
nxt.Switch.~, 476
nxt.UlraSonicSensor.~, 475
initial value, 254, 412
initialized
 Ros.Service.~, 428
InputStream, 318
 Global.~, 312
insert
 List.~, 330
insertBack
 List.~, 331
insertFront
 List.~, 331
insertUnique
 List.~, 331
inspect
 Object.~, 355
install dir, 180
installdir, 180
instances
 Lobby.~, 342
‘**--interactive**’, 192
Interface, 447
interface, 433
 Interface.AccelerationSensor, 438
 Interface.AudioIn, 433
 Interface.AudioOut, 434
 Interface.AutomaticGainControl, 499
 Interface.Battery, 434
 Interface.BlobDetector, 434
 Interface.Denoiser, 500
 Interface.DistanceSensor, 438
 Interface.GyroSensor, 439
 Interface.Identity, 435
 Interface.Laser, 439
 Interface.Led, 435
 Interface.LinearMotor, 437
 Interface.LinearSpeedMotor, 437
 Interface.Mobile, 435
 Interface.Motor, 437
 Interface.Network, 438
 Interface.RGBLed, 435
 Interface.RotationalMotor, 438
 Interface.RotationalSpeedMotor, 438
 Interface.Sensor, 438
 Interface.SpeechRecognizer, 440
 Interface.TemperatureSensor, 439

I
 Interface.TextToSpeech, 440
 Interface.TouchSensor, 439
 Interface.Tracker, 440
 Interface.VideoIn, 441
 '--interval=*interval*', 197
 Io, 571
 IoService, 319
 IoService, 319
 IP
 Interface.Network.~, 438
 ipAddress
 jazz.network.~, 491
 IRLeft, IRCenter, IRRight
 AXS1.~, 465
 isa
 Object.~, 355
 isConnected
 Socket.~, 387
 isdef
 Global.~, 312
 isDir
 Path.~, 365
 isInf
 Float.~, 302
 isNaN
 Float.~, 302
 isNil
 nil.~, 350
 Object.~, 355
 isProto
 Object.~, 355
 isReg
 Path.~, 365
 isVoid
 Object.~, 356
 void.~, 422
 isvoid
 Kernel1.~, 323
 isWindows
 System.Platform.~, 403

J
 '-j', 194, 199
 jazz.body.head, 488
 jazz.body.head.eye, 488
 jazz.body.laser, 497
 jazz.camera, 492
 jazz.network, 490
 Job, 321
 Global.~, 313
 '--jobs=*jobs*', 199
 join
 List.~, 331

Process.~, 371
 String.~, 391
 joint, 444
 '--jpeg=*factor*', 194

K
 '-k', 200
 kernel, 3
 Kernel1, 322
 Global.~, 313
 '--kernel=*dir*', 200
 key
 jazz.network.~, 491
 keys
 Dictionary.~, 278
 List.~, 331
 keyword, 203, 531
 keywords
 Binary.~, 259
 kill
 Process.~, 371
 kind
 Interface.Identity.~, 435
 System.Platform.~, 404
 knee, 443

L
 '-l', 200
 '-llib', 200
 '-Lpath', 200
 lang
 Interface.SpeechRecognizer.~, 440
 Interface.TextToSpeech.~, 440
 Laser, 439
 laserDistanceMax
 P3dx.body.laser.~, 508
 laserDistanceMin
 P3dx.body.laser.~, 508
 lastCaptureTimestamp
 P3dx.body.laser.~, 508
 lastModifiedDate
 Directory.~, 282
 File.~, 294
 latency
 uobjects.AlsaSpeaker.~, 498
 launch
 Timeout.~, 411
 Lazy, 324
 Global.~, 313
 lazy, 324
 leave
 Tag.~, 409
 leaveChargingDock

jazz.~, 494
 Led, 435
 left
 RMP.~, 511
 leftCoeff
 RMP.~, 512
 leftSpeed
 RMP.~, 511
 leftSpeedCoeff
 RMP.~, 512
 leg, 443
 length
 String.~, 391
 level
 Interface.AutomaticGainControl.~, 499
 Levels
 Logger.~, 346
 'libraries.stamp', 183
 '--library', 200
 license
 Lobby.~, 342
 lightLeft, lightCenter, lightRight
 AXS1.~, 465
 LightSensor, 476
 limits
 Float.~, 302
 line
 Position.~, 369
 LinearMotor, 437
 LinearSpeedMotor, 437
 lines
 Position.~, 369
 lip, 445
 List, 326
 Global.~, 313
 list, 165, 205
 listen
 Server.~, 384
 literals, 95
 load
 AX12.~, 465
 Interface.AutomaticGainControl.~, 499
 Interface.Denoiser.~, 500
 jazz.body.head.~, 489
 Loadable.~, 338
 P3dx.body.camera.~, 508
 P3dx.body.laser.~, 508
 P3dx.body.~, 506
 System.~, 397
 uobjects.EchoCanceller.~, 499
 Loadable, 337
 Global.~, 313
 loadFile

System.~, 397
 loadLibrary
 System.~, 397
 loadModule
 System.~, 397
 lobbies
 System.~, 397
 Lobby, 339
 Lobby, 339
 Global.~, 313
 lobby, 160, 339
 lobby
 Lobby.~, 342
 System.~, 397
 'local.u', 191
 localHost
 Socket.~, 387
 Localizer, 447
 localPort
 Socket.~, 387
 localSlotNames
 Object.~, 356
 locateSlot
 Object.~, 356
 Location, 343
 location
 Exception.~, 289
 StackFrame.~, 388
 locFailureEvent
 P3dx.planner.~, 509
 LOCK_CLASS, 57
 LOCK_FUNCTION, 57
 LOCK_FUNCTION_DROP, 57
 LOCK_FUNCTION_KEEP_ONE, 57
 LOCK_INSTANCE, 57
 LOCK_MODULE, 57
 LOCK_NONE, 57
 log
 Float.~, 302
 Logger.~, 347
 Math.~, 348
 Logger, 345
 Lookup
 Exception.~, 290
 loop
 closed-loop, 412, 416
 open-loop, 412, 416

M

'-m', 192, 197, 200
 makeCompactNames
 Component.~, 447
 makeServer

IoService.~, 320
 makeSocket
 IoService.~, 320
 manifest values, 95
 map
 List.~, 332
 mapFileName
 P3dx.planner.~, 509
 match
 Pattern.~, 367
 Regexp.~, 380
 matchAgainst
 Dictionary.~, 278
 List.~, 332
 Tuple.~, 418
 matches
 Regexp.~, 381
 MatchFailure
 Exception.~, 291
 matchPattern
 Pattern.~, 367
 Math, 347
 Global.~, 313
 max
 Float.limits.~, 306
 Float.~, 302
 List.~, 332
 Math.~, 348
 max_gain
 Interface.AutomaticGainControl.~, 499
 maxCallTime
 UConnection.~, 60
 maxExponent
 Float.limits.~, 306
 maxExponent10
 Float.limits.~, 306
 maxFunctionCallDepth
 Profile.~, 375
 maybeLoad
 System.~, 397
 meanCallTime
 UConnection.~, 60
 memoization, 324
 message, 98, 228
 call, 222
 message
 CallMessage.~, 263
 Exception.~, 289
 meta-variable, 531
 methods, 44
 methodToFunction
 Global.~, 313
 micro, 445
 uobjects.EchoCanceller.~, 499
 microsecond
 Date.~, 272
 min
 Float.limits.~, 306
 Float.~, 302
 List.~, 332
 Math.~, 348
 minCallTime
 UConnection.~, 60
 minExponent
 Float.limits.~, 306
 minExponent10
 Float.limits.~, 306
 minInterval
 UConnection.~, 60
 minute
 Date.~, 272
 Mobile, 435
 mode
 nxt.LightSensor.~, 476
 nxt.SoundSensor.~, 475
 model
 Interface.Identity.~, 435
 P3dx.~, 506
 ‘--module=*module*’, 192, 197
 month
 Date.~, 272
 Motor, 437
 mouth, 444
 moveInto
 Directory.~, 282
 File.~, 294
 moveTo
 P3dx.~, 506
 moving
 Interface.Mobile.~, 437
 jazz.~, 494
 mutable, 212
 Mutex, 350
 Mutex, 350
 Global.~, 313
N
 ‘-n’, 198
 name
 Channel.~, 265
 Enumeration.~, 286
 Interface.Identity.~, 435
 Job.~, 322
 P3dx.~, 506
 Process.~, 371
 Profile.Function.~, 376

R
 Ros.Service.~, 428
 Ros.Topic.~, 425
 Ros.~, 423
 StackFrame.~, 388
 nan
 Float.~, 302
 Math.~, 349
 Nao, 453
 NDEBUG
 System.~, 398
 neck, 444
 NegativeNumber
 Exception.~, 291
 netMask
 jazz.network.~, 491
 Network, 438
 new
 uobjects.AlsaMicrophone.~, 497
 uobjects.AlsaSpeaker.~, 498
 uobjects.EchoCanceller.~, 498
 nil, 350
 Global.~, 313
 '--no-header', 198
 nodes
 Ros.~, 423
 NonPositiveNumber
 Exception.~, 291
 noop
 Kernel1.~, 323
 not in, 216
 notations, 531
 notifyAccess
 UVar.~, 421
 notifyChange
 UVar.~, 421
 notifyChangeOwned
 UVar.~, 421
 now
 Date.~, 273
 null
 Channel.~, 265
 nxt.Battery, 476
 nxt.Beeper, 477
 nxt.Command, 477
 nxt.LightSensor, 476
 nxt.Servo, 475
 nxt.SoundSensor, 475
 nxt.Switch, 476
 nxt.UltraSonicSensor, 475

O
 '-o', 194, 198, 200
 Object, 309

Object, 351
 Global.~, 313
 object, 101, 224
 off
 Loadable.~, 338
 on
 Loadable.~, 338
 onConnect
 Ros.Topic.~, 426
 onDisconnect
 Lobby.~, 342
 Ros.Topic.~, 426
 onEnter
 Logger.~, 347
 onLeave
 Logger.~, 347
 onMessage
 Ros.Topic.~, 425
 onSubscribe
 Event.~, 287
 open
 Path.~, 365
 OpenLoop
 TrajectoryGenerator.~, 416
 Operating System, 3
 operator, 210
 arithmetics, 211
 bitwise, 214
 Boolean, 214
 comparison, 215
 subscript, 216
 Orderable, 361
 Global.~, 313
 orientation
 Interface.BlobDetector.~, 434
 '--output=*file*', 194, 198, 200
 OutputStream, 362
 Global.~, 313
 owned
 UVar.~, 421
 owned mode, 421

P
 p
 Semaphore.~, 382
 '-P', 192, 194, 197, 198, 200
 '-p', 194, 195, 200
 P3dx, 506
 P3dx.body, 506
 P3dx.body.battery, 509
 P3dx.body.camera, 508
 P3dx.body.laser, 508
 P3dx.body.odometry, 506

P3dx.body.sonar, 507
 P3dx.body.x, 508
 P3dx.body.yaw, 508
 P3dx.planner, 509
 '--package=*pkg*', 200
 PackageInfo
 System.~, 398
 pad
 FormatInfo.~, 307
 Pair, 363
 Global.~, 313
 pair, 363
 '--param-mk=*file*', 200
 parent
 Directory.~, 283
 Path, 363, 365
 Path, 363
 Global.~, 314
 pathPlanningEvent
 P3dx.planner.~, 509
 Pattern, 366
 Global.~, 314
 pattern
 FormatInfo.~, 308
 Pattern.~, 367
 pattern matching, 229
 payload, 135
 period, 398
 period
 System.~, 398
 '--period=*period*', 194
 persist
 Control.~, 270
 Global.~, 314
 PGain
 Interface.Motor.~, 437
 pi
 Float.~, 303
 Math.~, 349
 piece of code, 531
 ping
 Kernel1.~, 323
 pitch, 444
 Interface.TextToSpeech.~, 440
 Interface.Tracker.~, 440
 jazz.body.head.~, 488
 P3dx.body.camera.~, 508
 Platform
 System.~, 398
 play
 nxt.Beeper.~, 477
 playing
 Interface.AudioOut.~, 434
 plugin mode, 166
 '--plugin', 195
 poll
 IoService.~, 320
 Socket.~, 387
 pollFD
 uobjects.AlsaSpeaker.~, 498
 pollFor
 IoService.~, 320
 pollInterval
 Socket.~, 387
 pollOneFor
 IoService.~, 320
 port
 nxt.LightSensor.~, 476
 nxt.Servo.~, 475
 nxt.SoundSensor.~, 475
 nxt.Switch.~, 476
 nxt.UltraSonicSensor.~, 475
 Server.~, 384
 Socket.~, 387
 '--port-file=*file*', 192, 194, 197, 198
 '--port=*port*', 192, 194, 197, 198
 Position, 367
 Global.~, 314
 position
 Interface.LinearMotor.~, 437
 Interface.Mobile.~, 436
 jazz.~, 494
 P3dx.body.odometry.~, 506
 precision
 FormatInfo.~, 308
 prefix, 180
 prefix
 FormatInfo.~, 308
 '--prefix=*dir*', 200
 pressure
 Interface.TouchSensor.~, 439
 Primitive, 369
 Exception.~, 291
 Global.~, 314
 print
 Object.~, 356
 '--print-root', 191, 195, 196
 Process, 369
 Global.~, 314
 Profile, 372
 Global.~, 314
 profile
 System.~, 398
 Profile.Function, 375
 programName
 System.~, 398

prompt, 533
 properties, 121
properties
 Object.~, 356
 property, 120
protos, 164
 Object.~, 356
 prototype, 118, 226
PseudoLazy, 377
 Global.~, 314
publish
 PubSub.~, 377
 Ros.Topic.~, 426
PubSub, 377
 Global.~, 314
PubSub.Subscriber, 378
put
 OutputStream.~, 362

Q

‘-Q’, 197
 ‘-q’, 192, 199
quality
 Interface.VideoIn.~, 441
 ‘--quiet’, 192, 199
quit
 Lobby.~, 342
 ‘--quit’, 197
quote
 Channel.~, 265

R

r
 Interface.RGBLed.~, 435
 jazz.body.head.eye.~, 488
 ‘-R’, 194
 ‘-r’, 194, 195
radix
 Float.limits.~, 306
random
 Float.~, 303
 Math.~, 349
range
 List.~, 333
RangeIterable, 378
 Global.~, 314
rate
 Interface.Laser.~, 439
ratio
 Interface.BlobDetector.~, 434
readable
 Path.~, 365
reboot

System.~, 398
receive
 Lobby.~, 342
received
 Socket.~, 387
reconnect
 UConnection.~, 60
 ‘--reconstruct’, 194
Redefinition
 Exception.~, 291
redefinitionMode
 System.~, 398
reduce, 329
reduction
 Interface.Denoiser.~, 500
Regexp, 379
 Global.~, 314
release
 Semaphore.~, 382
relocatable, 151
remain
 Interface.AudioOut.~, 434
 Interface.Battery.~, 434
 uobjects.AlsaSpeaker.~, 498
remote mode, 166
 ‘--remote’, 195
remoteIP
 Lobby.~, 343
remove
 Directory.~, 283
 File.~, 294
 Group.~, 317
 List.~, 333
removeAll
 Directory.~, 283
removeBack
 List.~, 333
removeById
 List.~, 333
removeFront
 List.~, 334
removeLocalSlot
 Object.~, 356
removeNotifyAccess
 UVar.~, 421
removeNotifyChange
 UVar.~, 421
removeNotifyChangeOwned
 UVar.~, 421
removeProperty
 Object.~, 357
removeProto
 Object.~, 357

removeSlot
 Object.~, 357
 rename
 Directory.~, 283
 File.~, 295
 renunciationInterval
 jazz.~, 494
 replace
 String.~, 392
 reqStruct
 Ros.Service.~, 428
 request
 nxt.Command.~, 477
 Ros.Service.~, 428
 requireFile
 System.~, 399
 reset
 Kernel1.~, 323
 resetConnectionStats
 uobjects.~, 420
 resetStats
 System.~, 399
 UConnection.~, 60
 resolution
 Interface.Laser.~, 439
 Interface.VideoIn.~, 441
 jazz.body.laser.~, 497
 P3dx.body.laser.~, 508
 '--resolution=resolution', 194
 resStruct
 Ros.Service.~, 428
 reverse
 List.~, 334
 RGBLed, 435
 right
 RMP.~, 511
 rightCoeff
 RMP.~, 512
 rightSpeed
 RMP.~, 511
 rightSpeedCoeff
 RMP.~, 512
 RMP, 511
 RMP, 511
 robot, 442
 robotType
 P3dx.~, 506
 roll, 444
 root, 151
 Ros, 423
 Ros.Service, 428
 Ros.Topic, 424
 RotationalMotor, 438
 RotationalSpeedMotor, 438
 round
 Float.~, 303
 Math.~, 349
 routine, 228
 RTP, 65
 run
 Process.~, 371
 runningcommands
 Kernel1.~, 323
 runtime path, 156
 runTo
 Process.~, 372

S
 '-s', 192, 194, 195, 200
 safetyDistanceMax
 P3dx.body.~, 506
 safetyDistanceMin
 P3dx.body.~, 506
 save
 File.~, 295
 say
 Interface.TextToSpeech.~, 440
 '--scale=factor', 194
 Scheduling
 Exception.~, 291
 scientific notation, 203
 scope, 97, 217
 scope
 Tag.~, 409
 scopeTag, 406
 System.~, 399
 script
 Interface.TextToSpeech.~, 440
 'sdk-remote/libport/test-suite.log', 184
 'sdk-remote/src/tests/test-suite.log', 184
 search-path, 189
 searchFile
 System.~, 399
 searchPath
 System.~, 399
 UObject.~, 419
 uobjects.~, 420
 second
 Date.~, 273
 Pair.~, 363
 Triplet.~, 417
 seconds
 Duration.~, 285
 selfTime
 Profile.Function.~, 377
 selfTimePer

Profile.Function.~, 376
Semaphore, 381
 Global.~, 314
send
 Lobby.~, 343
 nxt.Command.~, 477
sender
 CallMessage.~, 263
Sensor, 438
seq
 Float.~, 303
 Kernell.~, 323
serial
 Interface.Identity.~, 435
 P3dx.~, 506
Server, 383
Server, 383
 Global.~, 314
Service
 Ros.~, 424
services
 Ros.~, 424
Servo, 475
set
 Dictionary.~, 278
setAbsolutePosition
 Interface.Mobile.~, 436
setConstSlot
 Object.~, 358
setenv
 System.~, 400
setProperty
 Group.~, 317
 Object.~, 358
setProtos
 Object.~, 358
setSideEffectFree
 Job.~, 322
setSlot
 Object.~, 359
setTrace
 uobjects.~, 420
setup
 uobjects.AlsaMicrophone.~, 498
 uobjects.AlsaSpeaker.~, 498
shared objects, 46
‘**--shared**’, 200
shiftedTime
 System.~, 400
shoulder, 443
shutdown
 System.~, 400
sign
Float.~, 303
Math.~, 349
signal
 Barrier.~, 257
signalAll
 Barrier.~, 257
Sin, 413
 TrajectoryGenerator.~, 416
sin
 Float.~, 303
 Math.~, 349
Singleton, 384
 Global.~, 314
singleton, 384
size
 Dictionary.~, 278
 Directory.~, 284
 Enumeration.~, 286
 File.~, 295
 Kernel1.~, 323
 List.~, 334
 String.~, 392
 Tuple.~, 418
sleep
 System.~, 400
slot, 101, 224
slotNames
 Object.~, 359
Smooth, 413
 TrajectoryGenerator.~, 416
Socket, 384
Socket, 384
 Global.~, 314
sockets
 Server.~, 384
sonar
 P3dx.body.~, 506
sort
 List.~, 334
SoundSensor, 475
soundVolume
 AXS1.~, 465
soundVolumeMax
 AXS1.~, 465
source directory, 180
spawn
 System.~, 400
speaker, 445
 uobjects.EchoCanceller.~, 499
speakout
 uobjects.EchoCanceller.~, 499
spec
 FormatInfo.~, 308

speech, 445
 SpeechRecognizer, 440
 Speed, 414
 speed
 AX12.~, 464
 Interface.GyroSensor.~, 439
 Interface.LinearSpeedMotor.~, 437
 Interface.RotationalSpeedMotor.~, 438
 Interface.TextToSpeech.~, 440
 nxt.Servo.~, 475
 P3dx.body.x.~, 508
 P3dx.body.yaw.~, 508
 SpeedAdaptive
 TrajectoryGenerator.~, 416
 spine, 445
 split
 String.~, 392
 Spykee, 513
 sqr
 Float.~, 303
 Math.~, 349
 sqrt
 Float.~, 303
 Math.~, 349
 srandom
 Float.~, 303
 Math.~, 349
 srmdir, 180
 stack size, 192
 '--stack-size=size', 192
 StackFrame, 387
 '--start', 195
 started?
 jazz.~, 494
 stats
 System.~, 401
 status
 Job.~, 322
 Process.~, 372
 stderr
 Process.~, 372
 stdin
 Process.~, 372
 stdout
 Process.~, 372
 stop, 404
 stop
 Interface.Mobile.~, 436
 jazz.~, 494
 P3dx.~, 506
 Tag.~, 409
 Stream, 388
 strict
 Kernel1.~, 324
 String, 389
 Global.~, 314
 string, 206, 365, 389, 531
 strlen
 Kernel1.~, 324
 structural pattern matching, 229
 structure
 Ros.Topic.~, 425
 structure tree, 429
 submit
 jazz.network.~, 491
 subscribe
 PubSub.~, 377
 Ros.Topic.~, 425
 Subscriber
 PubSub.~, 377
 subscriberCount
 Ros.Topic.~, 425
 subscribers
 PubSub.~, 377
 subset
 List.~, 334
 Switch, 476
 syncEmit
 Event.~, 287
 synchronous, 49
 syncline, 202
 syncTrigger
 Event.~, 287
 syncWrite
 Socket.~, 387
 Syntax
 Exception.~, 291
 System, 394
 Global.~, 314
 system
 System.~, 401
 System.PackageInfo, 402
 System.Platform, 403

T

Tag, 404
 Global.~, 314
 tag, 164, 404
 taglist
 Kernel1.~, 324
 tags
 Job.~, 322
 Tag.~, 409
 tail, 444
 List.~, 335
 tan

Float. \sim , 304
 Math. \sim , 349
 target, 98
 target
 CallMessage. \sim , 263
 target value, 254, 412
 temperature
 Interface.TemperatureSensor. \sim , 439
 TemperatureSensor, 439
 terminate
 Job. \sim , 322
 ‘tests/test-suite.log’, 184
 TextToSpeech, 440
 thanks
 Lobby. \sim , 343
 third
 Triplet. \sim , 417
 thread-safety, 165
 threshold
 Interface.BlobDetector. \sim , 434
 Time, 414
 TrajectoryGenerator. \sim , 416
 time
 System. \sim , 402
 TimeAdaptive
 TrajectoryGenerator. \sim , 416
 Timeout, 410
 Global. \sim , 314
 timeReference
 System. \sim , 402
 times
 Float. \sim , 304
 timeShift
 Job. \sim , 322
 timestamp
 Date. \sim , 273
 toAscii
 String. \sim , 392
 toe, 444
 toggle
 Loadable. \sim , 338
 toLower
 String. \sim , 393
 Topic
 Ros. \sim , 424
 topics
 Ros. \sim , 424
 topLevel
 Channel. \sim , 265
 torque
 AX12. \sim , 465
 Interface.RotationalMotor. \sim , 438
 torso, 445
 totalCalls
 Profile. \sim , 375
 totalTime
 Profile. \sim , 375
 touch, 445
 TouchSensor, 439
 toUpper
 String. \sim , 393
 trace
 Logger. \sim , 347
 Traceable, 411
 Tracker, 440
 trajectory, 254
 TrajectoryGenerator, 412
 Global. \sim , 314
 trigger
 Event. \sim , 287
 triple, 416
 Triplet, 416
 Global. \sim , 315
 triplet, 416
 true, 259
 Global. \sim , 315
 trunc
 Float. \sim , 304
 Math. \sim , 349
 Tuple, 417
 Global. \sim , 315
 tuple, 417
 tuples, 207
 turn
 Interface.Mobile. \sim , 436
 Interface.RotationalMotor. \sim , 438
 jazz. \sim , 494
 P3dx. \sim , 506
 turnAbsolute
 jazz. \sim , 495
 Type
 Exception. \sim , 292
 type
 Object. \sim , 359

U

UConnection, 60
 UConnection, 60
 uid
 Object. \sim , 359
 UltraSonicSensor, 475
 unacceptVoid
 Object. \sim , 359
 void. \sim , 422
 unadvertise
 Ros.Topic. \sim , 427

unblock, 405
 unblock
 Tag. \sim , 410
 undefall
 Kernel1. \sim , 324
 UnexpectedVoid
 Exception. \sim , 292
 unfreeze
 Tag. \sim , 410
 unreachable
 Interface.Mobile. \sim , 437
 unsetenv
 System. \sim , 402
 unstrict
 Kernel1. \sim , 324
 unsubscribe
 PubSub. \sim , 377
 Ros.Topic. \sim , 425
 UObject, 3, 165
 UObject, 419
 Global. \sim , 315
 uobjects, 419
 Global. \sim , 315
 uobjects.AlsaMicrophone, 497
 uobjects.AlsaSpeaker, 498
 uobjects.EchoCanceller, 498
 updateSlot
 Group. \sim , 317
 Object. \sim , 360
 uppercase
 FormatInfo. \sim , 308
 urbi, 35
 Urbi Runtime, 3
 urbi-launch, 35
 urbi-root, 151, 190
 ‘URBI.INI’, 191
 ‘urbi.stamp’, 183
 URBI_ACCEPT_BINARY_MISMATCH, 182
 URBI_CHECK_MODE, 182
 URBI_DESUGAR, 182
 URBI_DOC, 182
 URBI_IGNORE_URBI_U, 182
 URBI_INTERACTIVE, 182
 URBI_LAUNCH, 182
 URBI_NO_ICE_CATCHER, 182
 URBI_PARSER, 182
 URBI_PATH, 190
 URBI_REPORT, 182
 URBI_ROOT, 190
 URBI_ROOT_LIBname, 182
 URBI_SCANNER, 182
 URBI_SHARE, 182
 URBI_TEXT_MODE, 182, 190

URBI_TOLEVEL, 183
 URBI_UOBJECT_PATH, 191
 URBI_UVAR_HOOK_CHANGED, 191
 us
 Date. \sim , 273
 uservars
 Kernel1. \sim , 324
 UTF-8, 201
 UValue, 420
 Global. \sim , 315
 UVar, 166
 UVar, 420
 Global. \sim , 315

V

v
 Semaphore. \sim , 383
 ‘-V’, 199
 ‘-v’, 199
 val
 AX12. \sim , 464
 Interface.AudioIn. \sim , 433
 Interface.AudioOut. \sim , 434
 Interface.Laser. \sim , 439
 Interface.Led. \sim , 435
 Interface.Motor. \sim , 437
 Interface.Sensor. \sim , 438
 Interface.VideoIn. \sim , 441
 jazz.body.head. \sim , 490
 jazz.body.laser. \sim , 497
 jazz.camera. \sim , 492
 nxt.Battery. \sim , 476
 nxt.LightSensor. \sim , 476
 nxt.Servo. \sim , 475
 nxt.SoundSensor. \sim , 476
 nxt.Switch. \sim , 476
 nxt.UltraSonicSensor. \sim , 475
 P3dx.body.laser. \sim , 508
 uobjects.AlsaMicrophone. \sim , 498
 uobjects.AlsaSpeaker. \sim , 498
 uobjects.EchoCanceller. \sim , 499

value
 Lazy. \sim , 326
 values
 Enumeration. \sim , 286
 variable, 97, 531
 local, 219
 variadic, 216, 223
 vars
 Kernel1. \sim , 324
 ‘--verbose’, 199
 version
 System. \sim , 402

'--version', 191, 194–199
VideoIn, 441
visible
 Interface.BlobDetector.~, 435
voice, 445
 Interface.TextToSpeech.~, 440
voicexml
 Interface.TextToSpeech.~, 440
void, 422
 Global.~, 315
voltage
 Interface.Battery.~, 434
 P3dx.body.battery.~, 509
volume
 Interface.AudioOut.~, 434
VPATH, 199

W

'-w', 192
wait
 Barrier.~, 258
waitForChanges
 Job.~, 322
waitForTermination
 Job.~, 322
wall
 Global.~, 315
 Lobby.~, 343
wallClockTime
 Profile.~, 375
warn
 Global.~, 315
 Logger.~, 347
warning
 Channel.~, 265
watchdog
 Interface.Mobile.~, 436
watchdogInterval
 Interface.Mobile.~, 436
 jazz.~, 495
wb
 Interface.VideoIn.~, 441
wheel, 443
wheel[left].speed
 P3dx.body.~, 506
wheel[right].speed
 P3dx.body.~, 506
width
 FormatInfo.~, 308
 Interface.VideoIn.~, 441
 jazz.camera.~, 493
wrist, 443
writable

Path.~, 365
write
 Lobby.~, 343
 Socket.~, 387

X

x, 444
 Interface.BlobDetector.~, 435
 P3dx.body.odometry.~, 507

xfov
 Interface.VideoIn.~, 441

xSpeed
 jazz.~, 495

Y

y, 444
 Interface.BlobDetector.~, 435
 P3dx.body.odometry.~, 507

yaw, 444
 Interface.Tracker.~, 440
 jazz.body.head.~, 488
 P3dx.body.camera.~, 508
 P3dx.body.odometry.~, 507
 RMP.~, 511
yawCoeff
 RMP.~, 512

yawSpeed
 jazz.~, 495
 RMP.~, 511

yawSpeedCoeff
 RMP.~, 512

year
 Date.~, 273

yfov
 Interface.VideoIn.~, 441

yields
 Profile.~, 375

Z

z, 445
 P3dx.body.odometry.~, 507

zip
 List.~, 335

zoom
 P3dx.body.camera.~, 508