The Orocos User's Manual

Open RObot COntrol Software

1.0.4

The Orocos User's Manual : *Open RObot COntrol Software* : 1.0.4

Abstract

This document gives an introduction to the Orocos [http://www.orocos.org] *(Open RObot COntrol Software)* project. It contains a high-level overview of what Orocos (aims to) offer and the installation manual.

Orocos Version 1.0.4.

Table of Contents

List of Figures

List of Tables

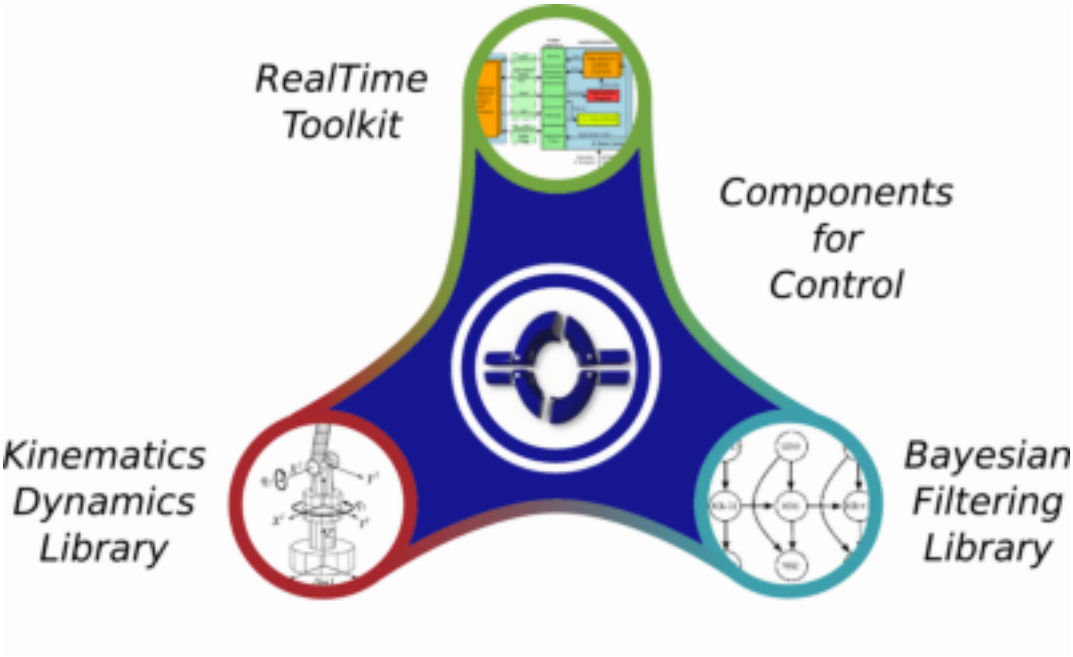
List of Examples

Chapter 1. Orocos Overview

This document gives an application oriented overview of Orocos [http://www.orocos.org], the *Open RObot COntrol Software* project.

1. What is Orocos?

"Orocos" is the acronym of the *Open Robot Control Software [http://www.orocos.org]* project. The project's aim is to develop a general-purpose, free software, and modular *framework* for *robot*and *machine control*. The Orocos project supports 4 C++ libraries: the Real-Time Toolkit, the Kinematics and Dynamics Library, the Bayesian Filtering Library and the Orocos Component Library.



- The Orocos Real-Time Toolkit (RTT) is not an application in itself, but it provides the infrastructure and the functionalities to build robotics applications in C++. The emphasis is on *real-time*, *online interactive* and *component based* applications.

- The Orocos Components Library (OCL) provides some ready to use control components. Both Component management and Components for control and hardware access are available.

- The Orocos Kinematics and Dynamics Library (KDL) is a C++ library which allows to calculate kinematic chains in real-time.

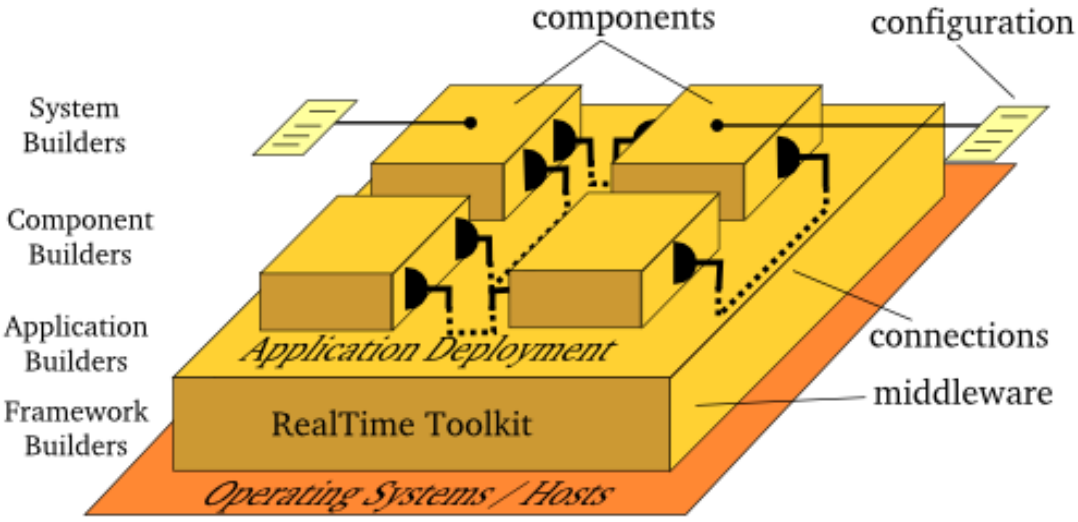
- The Orocos Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes' rule, such as (Extended) Kalman Filters, Particle Filters (Sequential Monte methods), etc.

Orocos is a free software project, hence its code and documentation are released under Free Software licenses.

Your feedback and suggestions are greatly appreciated. Please, use the project's mailinglist [http://lists.mech.kuleuven.be/mailman/listinfo/orocos] for this purpose.

2. Target audience

Robotics or machine control in general is a very broad field, and many roboticists are pursuing quite different goals, dealing with different levels of complexity, real-time control constraints, application areas, user interaction, etc. So, because the robotics community is not homogeneous, Orocos targets four different categories of "Users" (or, in the first place, "Developers"):



1. *Framework Builders.*

   These developers do not work on any specific application, but they provide the infrastructure code to support applications. This level of supporting code is most often neglected in robot software projects, because in the (rather limited) scope of each individual project, putting a lot of effort in a generic support platform is often considered to be "overkill", or even not taken into consideration at all. However, because of the large scope of the Orocos project, the supporting code (the "Framework") gets a lot of attention. The hope is, of course, that this work will pay of by facilitating the developments for the other "Builders". The RTT, KDL and BFL are created by Framework builders

2. *Component Builders.*

   Components provide a "service" within an application. Using the infrastructure of the framework, a Component Builder describes the interface of a service and provides one or more implementations. For example a Kinematics Component may be designed as such that it can "serve" different kinematic architectures. Other examples are Components to hardware devices, Components for diagnostics, safety or simulation. The OCL is created by Component Builders.

3. *Application Builders.*

   These developers use the Orocos' Framework and Components, and integrate them into one particular application. That means that they create a specific, application-dependent *architecture*: Components are connected and configured as such that they form an application.

4. *End Users.*

   These people use the products of the Application Builders to program and run their particular tasks.

End Users do not directly belong to the target audience of the Orocos project, because Orocos concentrates on the common *framework*, independent of any application architecture. Serving the needs of the End Users is left to (commercial and non-commercial) Application Builders.

3. Building Orocos Applications

Orocos applications are composed of software components, which form an application specific network. When using Orocos, you can choose to use predefined components, contributed by the community, or build your own component, using the Orocos Real-Time Toolkit. This section introduces both ways of building applications.

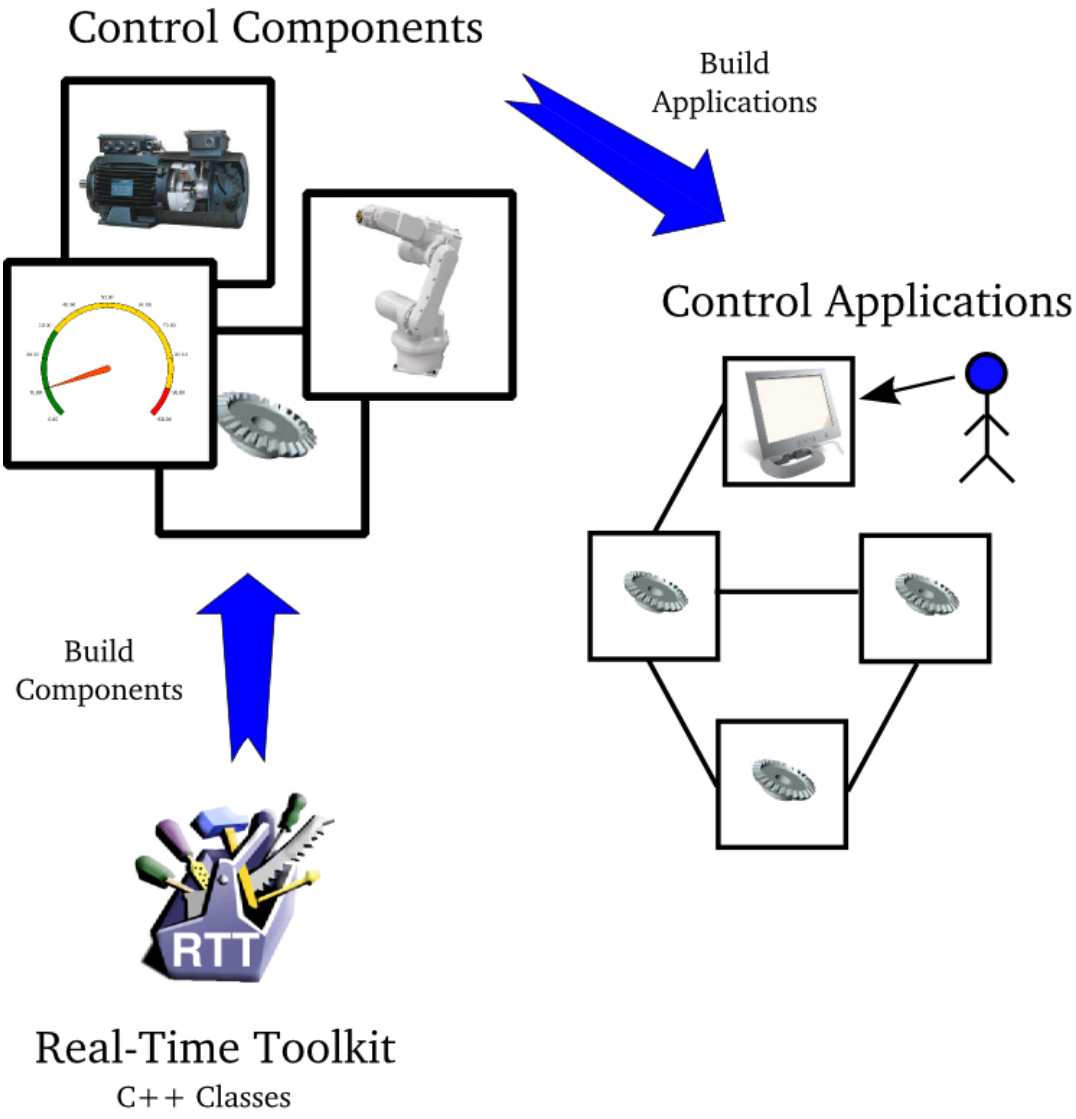


Figure 1.1. Orocos Real-Time Toolkit

3.1. Application Templates

An "Application Template" is a set of components that work well together. For example, the application template for motion control contains components for path planning, position control, hardware access and data reporting. The components are choosen as such that their interfaces are compatible.

An application template should be so simple that any Orocos user can pick one and modify it, hence it is the first thing a new user will encounter. An application template should be explainable on one page with one figure explaining the architecture.

Note

An application template has no relation to 'C++' templates.

3.2. Control Components

Applications are constructed using the Orocos "Control Component". A distributable entity which has a control oriented interface.

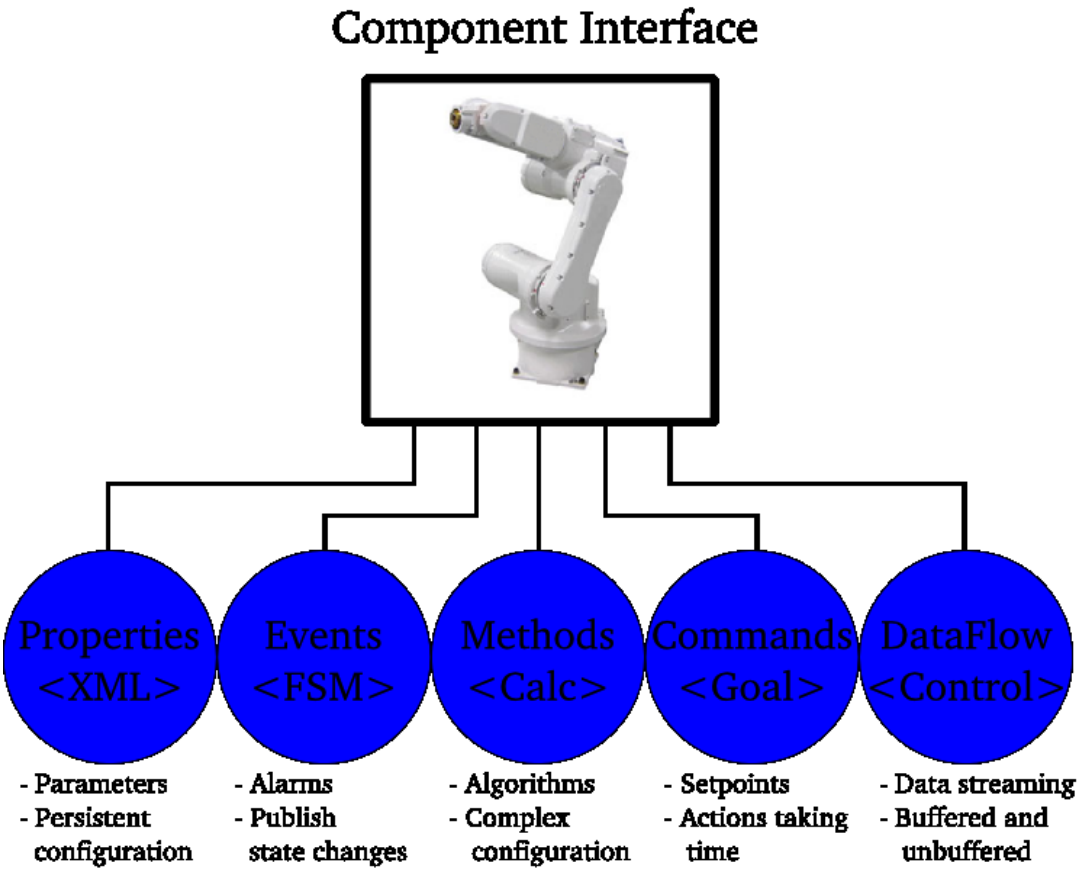


Figure 1.2. Orocos Control Component Interface

A single component may be well capable of controlling a whole machine, or is just a 'small' part in a whole network of components, for example an interpolator or kinematic component. The components are built with the "Real-Time Toolkit" and optionally make use of any other library (like a vision or kinematics toolkit). Most users will interface components through their (XML) properties or command/method interface in order to configure their applications.

There are five distinct ways in which an Orocos component can be interfaced: through its properties, events, methods, commands and data flow ports (Figure 1.2, “ Orocos Control Component Interface ”). These are all optional interfaces. The purpose and use of these interface 'types' is documented in the Orocos Component Builder's Manual. Each component documents its interface as well. To get a grip on what these interfaces mean, here are some fictuous component interfaces for a 'Robot' Component:

- *Data-Flow Ports:* Are a thread-safe data transport mechanism to communicate

buffered or un-buffered data between components. For example: "JointSetpoints", "EndEffectorFrame", "FeedForward",...

- *Properties:* Are run-time modifyable parameters, stored in XML files. For example: "Kinematic Algorithm", "Control Parameters", "Homing Position", "ToolType",...
- *Methods:* Are callable by other components to 'calculate' a result immediately, just like a 'C' function. For example: "getTrackingError()", "openGripper()", "writeData("filename")", "isMoving()", ...
- *Commands:* Are 'sent' by other components to instruct the receiver to 'reach a goal' For example: "moveTo(pos, velocity)", "home()",... A command cannot, in general, be completely executed instantaneously, so the caller should not block and wait for its completion. But the Command object offers all functionalities to let the caller know about the progress in the execution of the command.
- *Events:* Allows functions to be executed when a change in the system occurs. For example: "Position Reached", "Emergency Stop", "Object Grasped",...

Besides defining the above component communication mechanisms, Orocos allows the Component or Application Builder to write hierarchical state machines which use these primitives. This is the Orocos way of defining your application specific logic. State machines can be (un-)loaded at runtime in any component.

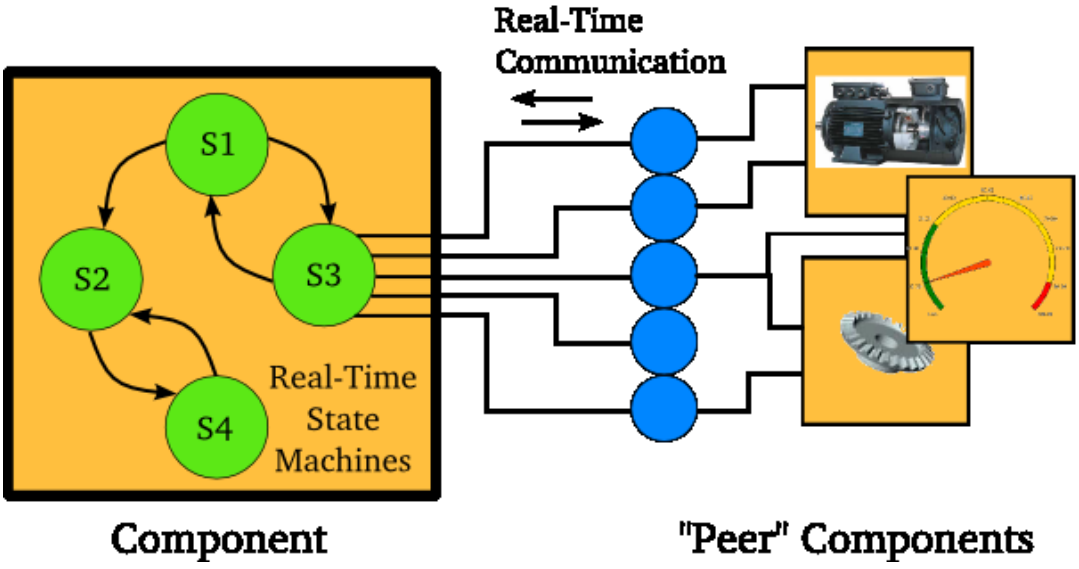


Figure 1.3. Orocos Control Component State Machines.

4. Related 'Orocos' Projects

The Orocos project spawned a couple of largely independent software projects. The documentation you are reading is about the Real-Time Control Software located on

the Orocos.org webpage. The other *not real-time* projects are :

- At KTH Stockholm, several releases have been made for component-based robotic systems, and the project has been renamed to Orca [http://orca-robotics.sourceforge.net/].
- Although not a project funded partner, the FH Ulm maintains Free CORBA communication patterns for modular robotics : Orocos::SmartSoft [http://www.rz.fh-ulm.de/~cschlege/orocos/].

This documentation is targetted at industrial robotics and real-time control.

Chapter 2. Installing Orocos

This document explains how the Real-Time Toolkit of Orocos [http://www.orocos.org], the *Open RObot COntrol Software* project must be installed and configured.

1. Setting up your first Orocos-RTT source tree

1.1. Introduction

This sections explains the supported Orocos targets and the Orocos versioning scheme.

1.1.1. Supported platforms (targets)

The Orocos Real-Time Toolkit was designed with portability in mind. Currently, we support RTAI/LXRT (http://www.rtai.org), GNU/Linux userspace, Xenomai (Xenomai.org [http://www.xenomai.org]). So, you can first write your software as a normal program, using the framework for testing and debugging purposes in plain userspace Linux and recompile later to another target.

1.1.2. The versioning scheme

Orocos uses the well-known even/stable uneven/unstable version numbering scheme, just as the Linux kernel and many other projects. A particular version is represented by three numbers separated by dots. An *even* middle number indicates a *stable* version. For example :

- 1.1.4 : Release 1, unstable (1), revision 4.
- 1.2.1 : Release 1, stable (2), revision 1.

This numbering allows to develop and release two kinds of versions, where the unstable version is mainly for testing new features and designs and the stable version is for people wanting to run a reliable system.

1.1.3. Dependencies on other Libraries

Before you install the RTT, verify that you have the following software installed on your platform :

Table 2.1. Build System Requirements

Program / Library	***Minimum* Version**	**Description**
TCL	8.0	tcl.tk [http://www.tcl.tk]
Python	2.2	Python.org [http://www.python.org]
GNU gcc / g++ Compilers	3.3.0 / 3.4.0	gcc.gnu.org [http://gcc.gnu.org] Orocos builds with the GCC 4.x series as well. For LXRT and Xenomai: compile these programs and their Linux kernel with GCC 3.x compilers and not with GCC 4.x.
pkg-config	1.0	Orocos uses this program to find out how to find other libraries on the system.
ecosconfig	2.0	ecoscentric [http://www.ecoscentric.com/devzone/configtool.shtml] For i386 and x86_64 (AMD64) platforms, this executable is packaged with Orocos. For other platforms, see the link. The 'configtool' sister program is optional.

Table 2.2. Run Requirements

Program / Library	***Minimum* Version**	**Description**
Boost C++ Libraries	0.32.0 (0.33.0 Recommended!)	Boost.org [http://www.boost.org] Version 0.33.0 has a very efficient (time/space) lock-free smart pointer implementation which is used by Orocos.
Xerces C++ Parser	2.1 (Optional)	Xerces website [http://xml.apache.org/xerces-c/] Versions 2.1 until 2.6 are known to work. If not found, an internal XML parser is used.
CppUnit Library	1.9.6 (Optional)	CppUnit website. [http://cppunit.sourceforge.net/cgi-bin/moin.cgi] Only needed if you want to run

Program / Library	***Minimum* Version**	**Description**
		the Orocos tests.
Xenomai	2.0.0 (2.2 Recommended)	Xenomai.org [http://www.xenomai.org] Extends the Linux kernel with a real-time scheduler.
RTAI	3.x	RTAI.org [http://www.rtai.org] Extends the Linux kernel with a real-time scheduler.

You'll require the following software to build the documentation. However, all documentation can be downloaded as a single archive from the RTT website [http://www.orocos.org/rtt].

Table 2.3. Doc Requirements (Optional)

Program / Library	***Minimum* Version**	**Description**
xsltproc	1.1.15	Install an XML processor [http://www.sagehill.net/docbookxsl/InstallingAProcessor.html]
docbook-xml	4.4	Docbook website. [http://www.docbook.org] The manuals are written in the XML format defined by the Docbook standard.
SUN JIMI	1.0	Docbook website. [http://www.docbook.org] Required to show PNG images in the generated PDF files.
Doxygen	1.3	Doxygen website. [http://www.doxygen.org] Required to generate/extract the API documentation.
Dia	0.91	Dia website. [http://www.gnome.org/projects/dia/] Required to generate images.
inkscape	0.42	Inkscape website. [http://www.inkscape.org] Required to generate images.

These packages are provided by most Linux distributions. Take also a look on the Orocos.org download page for the latest information.

1.2. Required Build Tools

The Real-Time Toolkit uses a less common system for distribution and configuration. It is taken from the eCos operating system, but, apart from the name, has nothing to do with it when used with Orocos.

1.2.1. Build and Configuration Tools

The tool you will need is **ecosconfig** In Debian, you can use the official Debian version using

```
apt-get install ecosconfig
```

If this does not work for you, Orocos will use the ready-to-use tools in Orocos' tools/bin subdirectory (i386 only). The build system will try to locate the tools in that directory and use them if present.

If the build tools cause problems, consult the eCos Configuration Tool Version 2 [http://www.ecoscentric.com/devzone/configtool.shtml] webpage for source and binary downloads.

Note

The optional **configtool** is a GUI in which users can configure some build options. It uses the libwxgtk2.4 library. The required **ecosconfig** is a commandline program, used by the make system.

1.2.2. Quick Installation Instructions

Download the orocos-rtt-1.0.4-src.tar.bz2 package from the Orocos webpage.

Extract it using :

```
tar -xvjf orocos-rtt-1.0.4-src.tar.bz2
```

Then proceed as in:

```
mkdir orocos-rtt-1.0.4/build
cd orocos-rtt-1.0.4/build
../configure --with-<target>
make rtt
make check
make install
```

Where <target> is one of listed in **../configure --help**. (currently 'gnulinux', 'lxrt' or 'xenomai'). When none is specified, 'gnulinux' is used.

Note

The **../configure** script must be rerun after you installed missing libraries (like Boost, ...).

1.2.3. Building The Real-Time Toolkit (RTT)

The RTT can be compiled in three flavours:

```
make rtt
make rtt-corba
make rtt-embedded
```

builds the 'standard', 'embedded' or 'corba' version. The embedded version is standard without scripting, the corba version is standard with Corba interfaces.

Note

Each one of these targets resets your configuration and starts building from scratch. Use plain **make** to continue building the current target.

1.2.4. Build results

The **make** command will have made a directory packages where the building takes place. The results of the build are in the packages/install directory. You will find the header files and a library called liborocos-rtt.a. These files allow you to build applications with Orocos.

The **make docs** and **make doxy-dist** (both in 'build') commands build manuals and API documentation in the build/doc directory.

1.2.5. Using Orocos concurrently for multiple targets

When you want to build for another target or operating system, create a new build-<target> directory and simply re-invoke ../configure --with-<target> from that build directory.

If this step fails, it means that you have not everything installed which is needed for a basic Orocos build. Most users don't have the Boost library (libboost-dev or libboost-devel) installed. Please install this package from the binary or source package repository of your Linux distribution, or download and install it from the Boost project. [http://www.boost.org] As soon as the configure step succeeds, all the rest will succeed too. Please use the mailinglist at <orocos-dev@lists.mech.kuleuven.be> for support questions.

1.3. Installing an Orocos Build

Orocos can optionally (*but recommended*) be installed on your system with

```
make install
```

The default directory is /usr/local/orocos, but can be changed with the --with-prefix option :

```
../configure --with-prefix=/opt/other/
```

If you choose not to install Orocos, you can find the build's result in the build/packages/install/[lib/include] directory.

2. Configuration Details

2.1. Configuring the RTT

In your build directory, run **../configure** or **../configure --with-gnulinux** to select the os/gnulinux target. Alternative targets are:

- --with-lxrt[=/usr/realtime] --with-linux[=/patched/linux/tree]
- --with-xenomai[=/usr/realtime] --with-linux[=/patched/linux/tree]

The latter targets require the presence of the --with-linux=/path/to/linux option since these targets require Linux headers during the Orocos build. To use the LibC Kernel headers in /usr/include/linux, specify --with-linux=/usr. Watch carefully the output to find any errors.

2.2. General Build Compiler Flags

You must set the compiler flags during the 'configure' step. For example:

```
../configure CXXFLAGS="-g -O0" CFLAGS="-g -O0" --with-gnulinux
```

if you want to enable debugging or change the optimisation flags.

For convenience, the configure script can also be invoked with the --enable-debug option, which installs the same flags as above.

2.3. Building for RTAI / LXRT

Read first the 'Getting Started' section from this page [http://people.mech.kuleuven.be/~psoetens/portingtolxrt.html] if you are not familiar with RTAI installation

Orocos has been tested with RTAI 3.0, 3.1, 3.2 and 3.3. You can obtain it from the RTAI home page [http://www.aero.polimi.it/projects/rtai/]. Read The README.* files in the rtai directory for detailed instructions. First, you need to patch your Linux kernel with the RTAI patch. A patch per kernel version can be found in the

rtai-core/arch/i386/patches directory. You should apply the hal12-X.Y.Z.patch (or later) for RTAI to a clean Linux-X.Y.Z kernel. We refer to the RTAI installation instructions for more details.

Next do **make menuconfig**

2.3.1. RTAI settings

RTAI comes with documentation for configuration and installation. In the configuration process, make sure that you enable the following options (*in addition to options you feel you need for your application*) :

- General -> 'Enable extended configuration mode'
- Core System -> Native RTAI schedulers > Scheduler options -> 'Number of LXRT slots' ('500')
- Machine -> 'Enable FPU support'
- Core System -> Native RTAI schedulers > IPC support -> Semaphores, Fifos, Bits (or Events) and Mailboxes
- Add-ons -> 'Comedi Support over LXRT' (if you intend to use the Orocos Comedi Drivers)
- Core System -> Native RTAI schedulers > 'LXRT scheduler (kernel and user-space tasks)'

After configuring you must run 'make' and 'make install' in your RTAI directory: **make ; make install**

Note

In recent version of RTAI, some of these options may have changed in location or name.

Warning

The patched Linux kernel and RTAI must be compiled with a GCC 3.x compiler and not with a GCC 4.x compiler! Otherwise, your system may become unbootable or unstable.

After installation, RTAI can be found in /usr/realtime. You'll have to specify during the Orocos configure step the alternative if you chose so.

2.3.2. Loading RTAI with LXRT

LXRT is a all-in-one scheduler that works for kernel and userspace. So if you use this, you can still run kernel programs but have the ability to run realtime programs in userspace. Orocos provides you the libraries to build these programs. Make sure

that the following RTAI kernel modules are loaded

- rtai_sem
- rtai_lxrt
- rtai_hal
- adeos (not present in all RTAI versions)

For example, by executing as root: **modprobe rtai_lxrt; modprobe rtai_sem**.

For a more detailed description of what LXRT really is, you can read the LXRT HOWTO here [http://people.mech.kuleuven.be/~psoetens/lxrt/portingtolxrt.html]

2.3.3. Configuring Orocos for LXRT

In your build directory, run: **../configure --with-lxrt[=/usr/realtime] --with-linux[=/patched/linux/tree]** The LXRT target requires the presence of the --with-linux=/path/to/linux option since these targets require Linux headers during the Orocos build. To use the LibC Kernel headers in /usr/include/linux, specify --with-linux=/usr.

2.3.4. Compiling Applications with LXRT

Note

If your application uses the orocos-rtt.pc file (pkg-config format), it will automatically find the flags below.

Application which use LXRT as a target need special flags when being compiled and linked. Especially :

- Compiling : -I/usr/realtime/include

  This is the RTAI headers installation directory.

- Linking : -L/usr/realtime/lib -llxrt for dynamic (.so) linking OR add /usr/realtime/liblxrt.a for static (.a) linking.

Important

You might also need to add /usr/realtime/lib to the /etc/ld.so.conf file and rerun **ldconfig**, such that liblxrt.so can be found. This option is not needed if you configured RTAI with LXRT-static-inlining.

2.4. Building for Xenomai

Xenomai provides a real-time scheduler for Linux applications. It is similar to, and once developed with, RTAI/LXRT, but the project is now independent of RTAI. See the Xenomai home page [http://www.xenomai.org]. Xenomai requires a patch one needs to apply upon the Linux kernel. See the Xenomai installation manual. When applied, one needs to enable the ADEOS or IPIPE option during Linux kernel configuration.

Warning

The patched Linux kernel and Xenomai must be compiled with a GCC 3.x compiler and not with a GCC 4.x compiler! Otherwise, your system may become unbootable or unstable.

When the Linux kernel is built, do in the Xenomai directory: **make menuconfig; make; make install**. Consult the README files in the top Xenomai directory for further help with installation.

2.4.1. Xenomai settings

The default settings for Xenomai are fine for Orocos. The only option that requires to be set is the location of the patched Linux kernel. After configuring you can run 'make' and 'make install' in your Xenomai directory.

After installation, Xenomai can be found in /usr/realtime or /usr/xenomai. Orocos looks in the first directory for Xenomai. You'll have to specify during the Orocos configure step the alternative if you chose so.

2.4.2. Loading Xenomai 2.0

Orocos uses the native Xenomai API to address the real-time scheduler. The Xenomai kernel modules can be found in /usr/realtime/modules. Thus only the following kernel modules need to be loaded:

- xeno_hal.ko
- xeno_nucleus.ko
- xeno_native.ko

in that order. For example, by executing as root: **insmod xeno_hal.ko; insmod xeno_nucleus.ko; insmod xeno_native.ko**.

2.4.3. Loading Xenomai 2.1 and later

For these versions, it is recommended that Xenomai is built-in the kernel. As such, no kernel modules need to be loaded.

2.4.4. Configuring Orocos for Xenomai

In your build directory, run: **../configure --with-xenomai[=/usr/realtime] --with-linux[=/patched/linux/tree]** The Xenomai target requires the presence of the --with-linux=/path/to/linux option since these targets require Linux headers during the Orocos build. To use the LibC Kernel headers in /usr/include/linux, specify --with-linux=/usr.

2.4.5. Compiling Applications with Xenomai

Note

If your application uses the orocos-rtt.pc file (pkg-config format), it will automatically find the flags below.

Application which use Xenomai as a target need special flags when being compiled and linked. Especially :

- Compiling : -I/usr/realtime/include

  This is the Xenomai headers installation directory.

- Linking : -L/usr/realtime/lib -lnative for dynamic (.so) linking OR add /usr/realtime/libnative.a for static (.a) linking.

Important

You might also need to add /usr/realtime/lib to the /etc/ld.so.conf file and rerun **ldconfig**, such that libnative.so can be found.

2.5. Configuring for CORBA

Orocos is recommended to be used with 'The Ace Orb' or *TAO* version prepared by OCI (Object Computing Inc.). You can find the latest TAO version on OCI's TAO website [http://www.theaceorb.com]. Orocos was tested with OCI's TAO 1.3 and 1.4. The OCI version is more stable than the versions provided by the DOC group on the Real-time CORBA with TAO (The ACE ORB) website [http://www.cs.wustl.edu/~schmidt/TAO.html].

Note

Orocos requires the ACE, TAO and TAO-orbsvcs libraries and header files to be installed on your workstation and *that the ACE_ROOT and TAO_ROOT variables are set.*

2.5.1. TAO installation (Optional)

Note

If your distribution does not provide the TAO libraries, or you want to use the OCI version, you need to build manually. These instructions are for building on Linux. See the ACE and TAO installation manuals for building on your platform.

You need to make an ACE/TAO build on your workstation. Download the package here: OCI Download [http://www.theaceorb.com/downloads/1.4a/index.html]. Unpack the tar-ball, and enter ACE_wrappers. Then do: **export ACE_ROOT=$(pwd) export TAO_ROOT=$(pwd)/TAO** When using GNU/Linux, continue with: **ln -s ace/config-linux.h ace/config.h ln -s include/makeinclude/platform_linux.GNU include/makeinclude/platform_macros.GNU** Finally, type: **make cd TAO make cd orbsvcs make** This finishes your TAO build.

2.5.2. Configuring Orocos for TAO

Orocos will first try to detect your location of ACE and TAO using the ACE_ROOT and TAO_ROOT environment variables.

Alternatively, you may add a configure flag to tell Orocos where to find the ACE_wrappers directory:

```
../configure --with-ace=/path/to/ACE_wrappers
        --with-tao=/path/to/ACE_wrappers/TAO
```

Orocos then assumes that the ACE headers are then found in /path/to/ACE_wrappers/ace, the TAO headers are found in /path/to/ACE_wrappers/TAO/tao, the libraries are found in /path/to/ACE_wrappers/lib and the IDL compiler in /path/to/ACE_wrappers/bin.

If neither ACE_ROOT and TAO_ROOT is set and --with-ace and --with-tao is not used, Orocos is not built using CORBA. The --with flags override the environment variables.

Note

Debian based systems may use the flags: **--with-ace=/usr/share/ace --with-tao=/usr/share/ace/TAO**.

Fedora based systems may use the flags: **--with-ace --with-tao**.

2.5.3. IDL File Compilation

When Orocos detects the presence of TAO, it immediately generates the client and server files for all the Orocos IDL files during the configure step. Thus in order to re-generate these files, the configure script needs to be called again. When you do not make use of Orocos' Corba functionality, these files are not compiled nor used in any way by the Orocos build.

2.5.4. Application Development with TAO

Note

If your application uses the orocos-rtt.pc file (pkg-config format), it will automatically find the flags below.

Once you compile and link your application with Orocos and with the Corba functionality enabled, you must provide the correct include and link flags in your own Makefile if TAO and ACE are not installed in the default path. Then you must add:

- Compiling : -I/path/to/ACE_wrappers -I/path/to/ACE_wrappers/TAO

  This is the ACE build directory in case you use OCI's TAO packages. This option is not needed if you used your distribution's TAO installation, in that case, TAO is in the standard include path.

- Linking : -L/path/to/ACE_wrappers/lib -lTAO -lACE -lTAO_IDL_BE -lTAO_PortableServer -lTAO_CosNaming

  This is again the ACE build directory in case you use OCI's TAO packages. The *first* option is not needed if you used your distribution's TAO installation, in that case, TAO is in the standard library path.

Important

You also need to add /path/to/ACE_wrappers/lib to the /etc/ld.so.conf file and rerun **ldconfig**, such that these libraries can be found. Or you can before you start your application type

```
export LD_LIBRARY_PATH=/path/to/ACE_wrappers/lib
```

.

3. Getting Started with the Code

This Section provides a short overview of how to proceed next using the Orocos Real-Time Toolkit.

3.1. A quick test

You can issue a **make check** in the Orocos build directory.

To quickly test an Orocos application, you can download the examples from the webpage. They may require that additional libraries are installed, like the Orocos Components Library.

3.2. What about main() ?

The first question asked by many users is : How do I write a test program to see how it works?

Building a sample application with Orocos is quite simple, but some care must be taken in initialising the realtime environment. First of all, you need to provide a function int ORO_main(int argc, char** argv) {...}, defined in <rtt/os/main.h> which contains your program :

```
#include <rtt/os/main.h>

int ORO_main(int argc, char** argv)
{
  // Your code, do not use 'exit()', use 'return' to
  // allow Orocos to cleanup system resources.
}
```

If you link with the liborocos-rtt.a library, this function will be called after the run-time environment is set up. To put in other words, the Orocos library already contains a main() function which will call the user-defined ORO_main() function.

Example 2.1. A Makefile for an Orocos Application

You can then simply compile your program with a Makefile resembling this one :

```
OROPATH=/usr/local/orocos
CXXFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config
orocos-rtt --cflags`
LDFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config orocos-rtt
--libs`

all: myprogram.cpp
     g++ myprogram.cpp ${CXXFLAGS} ${LDFLAGS} -o myprogram
```

The flags must be extended with compile and link options for your particular configuration.

Important

The LDFLAGS option must be placed after the .cpp or .o files in the gcc command.

Note

Make sure you have read Section 2, “Configuration Details” for your target if you application has compilation or link errors (for example when using LXRT).

3.3. Header Files Overview

Table 2.4. Header Files

Header	Summary
rtt/*.hpp	The `Real-Time Toolkit' directory contains all the headers an Orocos user requires for building components.
rtt/os/*.h, rtt/os/*.hpp	Not intended for normal users. The os headers describe a limited set of OS primitives, like locking a mutex or creating a thread. Read the OS manual carefully before using these headers, they are mostly used internally in the corelib's implementation.
rtt/dev/*.h[pp]	C++ Headers of device interfaces
rtt/corba/*.hpp	C++ Headers for Corba support.

4. Expert configuration and embedded systems.

For embedded systems, the **make rtt-embedded** target should be built. This removes the scripting files from the build.

It is possible to shrink even more code size by using the **make configure_packages** command and disable exceptions and enable 'embedded operating system'. Next proceed with **make build** to build the adapted configuration.

Warning

Invoking **make rtt-embedded** removes any configuration changes you have made. Use **make build** (or **make cross**) to build a custom configured configuration. Use **make new_packages** to start with a clean configuration.

See the Developer's Manual [orocos-devel-manual.html] for using the configtool program which allows to select what gets built and which allows to tune build parameters for embedded systems.

5. Cross Compiling Orocos

This section lists some points of attention when cross-compiling Orocos.

5.1. Configuration

The following configure command is an example for supported cross compilation, using a Xenomai build:

```
$ cd build
$ ../configure --build=../config/config.guess \
--host=powerpc-405-linux-gnu \
--with-xenomai=/target-fs/usr/xenomai/ \
--with-linux=/target-fs/linuxppc_2_4_devel/ \
CXXFLAGS="-I/target-fs/usr/include/" \
CFLAGS="-I/target-fs/usr/include/" \
CC=/crosstool/bin/powerpc-405-linux-gnu-gcc \
CXX=/crosstool/bin/powerpc-405-linux-gnu-g++
```

when this finished succesfully, the 'make' command can be issued:

```
$ make
```

and a cross compilation build will run.

Additionally, you can:

- Start the configtool with "make configure_packages"
- Use the configtool's "Global Build Options" to set your compiler and include paths.

In case you wish to specify the compiler at 'make' time, use the following make target:

```
make cross CC=... CXX=...
```