

---

# The OROCOS Real-Time Toolkit Installation Guide

*Real-Time Toolkit Version 1.4.1*

Copyright © 2002,2003,2004,2005,2006,2007 Peter SoetensFMTC

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at <http://www.fsf.org/copyleft/fdl.html>.

	Revision History	
Revision 1.0.0	27 Okt 2006	ps
	Simplified build system.	
Revision 1.0.1	21 Nov 2006	ps
	Updated build/run/doc dependencies.	
Revision 1.1.0	13 Apr 2007	ps
	Rewritten for Orocos 1.2.0.	
Revision 1.2.1	02 June 2007	ps
	Minor clarifications.	
Revision 1.4.0	22 Nov 2007	ps
	Changes in the library name (-target) and .pc files	
Revision 1.4.1	12 Feb 2008	ps
	Added Debian/Ubuntu packages install instructions and updated Getting started/Makefile section.	

## Abstract

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.

## Table of Contents

1. Setting up your first Orocos source tree .....	2
1.1. Introduction .....	2
1.2. Installing on Debian Etch .....	3
1.3. Basic Real-Time Toolkit Installation .....	4
1.4. Installing an Orocos Build .....	6
2. Detailed Configuration using 'CMake' .....	7
2.1. Configuring the target Operating System .....	7
2.2. Setting Build Compiler Flags .....	7
2.3. Building for RTAI / LXRT .....	8
2.4. Building for Xenomai (version 2.2.0 or newer) .....	9
2.5. Configuring for CORBA .....	10

3. Getting Started with the Code .....	12
3.1. A quick test .....	12
3.2. What about main() ? .....	12
3.3. Building components and applications .....	13
3.4. Header Files Overview .....	14
4. Cross Compiling OrocOS .....	14

# 1. Setting up your first OrocOS source tree

## 1.1. Introduction

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

### 1.1.1. Supported platforms (targets)

OrocOS 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 Linux program, using the framework for testing and debugging purposes in plain user-space Linux and recompile later to another real-time target.

### 1.1.2. The versioning scheme

OrocOS uses the even/stable uneven/unstable version numbering scheme, just as the Linux kernel. 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 users wanting to run a reliable system.

### 1.1.3. Dependencies on other Libraries

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

**Table 1. Build Requirements**

Program / Library	Minimum Version	Description
Boost C++ Libraries	0.32.0 (0.33.0 or newer)	Boost.org [ht-

Program / Library	Minimum Version	Description
	Recommended!)	<a href="http://www.boost.org">tp://www.boost.org</a> ] Version 0.33.0 has a very efficient (time/space) lock-free smart pointer implementation which is used by Orocos.
GNU gcc / g++ Compilers	3.3.0	<a href="http://gcc.gnu.org">gcc.gnu.org</a> [ <a href="http://gcc.gnu.org">http://gcc.gnu.org</a> ] Orocos builds with the GCC 4.x series as well.
Xerces C++ Parser	2.1 (Optional)	Xerces website [ <a href="http://xml.apache.org/xerces-c/">http://xml.apache.org/xerces-c/</a> ] Versions 2.1 until 2.6 are known to work. If not found, an internal XML parser is used.
ACE & TAO	TAO 1.3 (Optional)	ACE & TAO website [ <a href="http://www.cs.wustl.edu/~schmidt/">http://www.cs.wustl.edu/~schmidt/</a> ] When you start your components in a networked environment, TAO can be used to set up communication between components.
CppUnit Library	1.9.6 (Optional)	CppUnit website. [ <a href="http://cppunit.sourceforge.net/cgi-bin/moin.cgi">http://cppunit.sourceforge.net/cgi-bin/moin.cgi</a> ] Only needed if you want to run the Orocos tests.

All these packages are provided by most Linux distributions. Take also a look on the Orocos.org RTT download [<http://www.orocos.org/rtt/source>] page for the latest information.

## 1.2. Installing on Debian Etch

The Orocos Real-Time Toolkit and Component Library have been prepared as Debian packages for Debian Etch. See below for instructions for building your own packages on other distributions, like Ubuntu.

For debian, copy/paste the following commands, and enter your password when asked:

```
wget -q -O - http://people.mech.kuleuven.be/~psoetens/gpg/key.gpg | sudo apt-key add -
sudo wget -q http://www.fmtc.be/sources.list.d/fmtc.list -O /etc/apt/sources.list.d/fmtc.list
```

These commands install the GPG key and the repository location of the Orocos packages.

Next type:

```
$ apt-get update
$ apt-get install liborocos-rtt-corba-gnulinix-dev
```

You can install Orocos for additional targets by replacing *gnulinix* by another target name (see above) if you want real-time performance. All targets can be installed at the same time.

For your application development, you'll most likely use the Orocos Component library as well:

```
$ apt-get install orocos-ocl-gnulinix-dev orocos-ocl-gnulinix-bin
```

Again, you may install additional targets.

We recommend using the `pkg-config` tool to discover the compilation flags required to compile your application with the RTT or OCL.

You can skip the rest of this manual and go to Section 3, “Getting Started with the Code” for compiling components and applications with the RTT.

If you're using Ubuntu, you need to source the packages and build them yourself, this is not such a hard task! Type from your '\$HOME/src' directory:

```
$ cd src
$ apt-get source orocos-rtt
$ apt-get build-dep orocos-rtt
$ apt-get install devscripts build-essential fakeroot dpatch
$ cd orocos-rtt-1.4.0
$ dpkg-buildpackage -rfakeroot -uc -us
$ cd ..
$ for i in *.deb; sudo dpkg -i $i; done
```

You can repeat the same process for `orocos-ocl`.

## 1.3. Basic Real-Time Toolkit Installation

The RTT uses the CMake [<http://www.cmake.org>] build system for configuring and building the library.

### 1.3.1. Orocos Build and Configuration Tools

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

```
apt-get install cmake
```

If this does not work for you, you can download `cmake` from the CMake homepage.

### 1.3.2. Quick Installation Instructions

Download the `orocos-rtt-1.4.1-src.tar.bz2` package from the Orocos webpage.

Extract it using :

```
tar -xvjf orocos-rtt-1.4.1-src.tar.bz2
```

Then proceed as in:

```
mkdir orocos-rtt-1.4.1/build
cd orocos-rtt-1.4.1/build
./configure --with-<target> [--prefix=/usr/local][--with-linux=/usr/src/linux]
make
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.
- --prefix specifies where to install the RTT.
- --with-linux is required for RTAI/LXRT and older Xenomai version (<2.2.0). It points to the source location of the RTAI/Xenomai patched Linux kernel.

See **configure --help** for a full list of options.



### Note

The **configure** script is a wrapper around the 'cmake' command and must be rerun after you installed missing libraries (like Boost, ...) before you can build the RTT.

## 1.3.3. Real-Time Toolkit Configuration

The RTT can be configured depending on you target. For embedded targets, the large scripting infrastructure and use of exceptions can be left out. When CORBA is available, an additional library is built which allows components to communicate over a network.

In order to configure the RTT in detail, you need to invoke the **ccmake** command:

```
cd orocos-rtt-1.4.1/build
ccmake ..
```

from your build directory. It will offer a configuration screen. The keys to use are 'arrows'/'enter' to modify a setting, 'c' to run a configuration check (may be required multiple times), 'g' to generate the makefiles. If an additional configuration check is required, the 'g' key can not be used and you must press again 'c' and examine the output.

## RTT with CORBA plugin

In order to enable CORBA a valid installation of TAO must be detected on your system and you must turn the `ENABLE_CORBA` option on (using `ccmake`). Enabling CORBA does not modify the RTT library, but builds and installs an additional library and headers.

Alternatively, when you use the configure wrapper, you can specify:

```
../configure --enable-corba
```

## Embedded RTT flavour

In order to run OrocOS applications on embedded systems, one can turn the `OS_EMBEDDED` option on. Next press 'c' again and additional options will be presented which allow you to select what part of the RTT is used. By default, the `OS_EMBEDDED` option already disables some 'fat' features. One can also choose to build the RTT as a static library (`BUILD_STATIC`).

Alternatively, when you use the configure wrapper, you can specify:

```
../configure --embedded
```

### 1.3.4. Build results

The `make` command will have created a `liborocos-rtt-<target>.so` library, and if CORBA is enabled a `liborocos-rtt-corba-<target>.so` library.

The `make docapi` and `make docpdf dohtml` (both in 'build') commands build API documentation and PDF/HTML documentation in the `build/doc` directory.

### 1.3.5. Building OrocOS for multiple targets

When you want to build for another target, 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. 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. Use the mailinglist at `<orocos-dev@lists.mech.kuleuven.be>` for support questions.

## 1.4. Installing an OrocOS Build

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

```
make install
```

The default directory is /usr/local, 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/src directory.

## 2. Detailed Configuration using 'CMake'

In order to start cmake configuration, in your build directory, run **ccmake ..** . Press 'c' (from 'c'onfigure), watch the output, press 'e' (from 'e'xit) and modify the new options. Repeat these steps until no errors are reported and the 'g' (from 'g'enerate) key can be pressed. This causes the makefiles to be generated which allow the library to be built.

### 2.1. Configuring the target Operating System

Move to the OROCOS\_TARGET, press enter and type on of the following supported targets (all in lowercase):

- gnulinux
- xenomai
- lxrt

The xenomai and lxrt targets require the presence of the LINUX\_SOURCE\_DIR option since these targets require Linux headers during the OrocOS build. To use the LibC Kernel headers in /usr/include/linux, specify /usr. Inspect the output to find any errors.



#### Note

From Xenomai version 2.2.0 on, Xenomai configuration does no longer require the --with-linux option.

### 2.2. Setting Build Compiler Flags

You can set the compiler flags using the CMAKE\_BUILD\_TYPE option. You may edit this field to contain:

- RTT (default)

- Release
- Debug
- RelWithDebInfo

## 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, 3.3, 3.4 and 3.5. 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 build instructions, as these depend on the RTAI version.

### 2.3.1. RTAI settings

RTAI comes with documentation for configuration and installation. During 'make menuconfig', 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' ('1000')
- 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 sudo make install**

After installation, RTAI can be found in /usr/realtime. You'll have to specify this directory in the RTAI\_INSTALL\_DIR option during 'ccmake'.

### 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` (depends on RTAI version)

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

### 2.3.3. Compiling Applications with LXRT

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 (version 2.2.0 or newer)



#### Note

For older Xenomai versions, consult the Xenomai README of that version.

Xenomai provides a real-time scheduler for Linux applications. See the Xenomai home page [<http://www.xenomai.org>]. Xenomai requires a patch one needs to apply upon the Linux kernel, using the `scripts/prepare-kernel.sh` script. See the Xenomai installation manual. When applied, one needs to enable the General Setup -> Interrupt Pipeline option during Linux kernel configuration and next the Real-Time Sub-system -> , Xenomai and Nucleus. Enable the Native skin, Semaphores, Mutexes and Memory Heap. Finally enable the Posix skin as well.

When the Linux kernel is built, do in the Xenomai directory: **./configure ; make; make install**.

You'll have to specify the install directory in the XENOMAI\_INSTALL\_DIR option during 'ccmake'.

## 2.4.1. Loading Xenomai

The RTT uses the native Xenomai API to address the real-time scheduler. The Xenomai kernel modules can be found in /usr/xenomai/modules. 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**.

## 2.4.2. Compiling Applications with Xenomai

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

- Compiling : **-I/usr/xenomai/include**

This is the Xenomai headers installation directory.

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



### Important

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

## 2.5. Configuring for CORBA

In case your application benefits from remote access over a network, the RTT can 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>]. The RTT was tested with OCI's TAO 1.3 and 1.4. The OCI version is preferred above 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** Configure ACE for Linux by doing: **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 the RTT for TAO

Orocos will first try to detect your location of ACE and TAO using the ACE\_ROOT and TAO\_ROOT variables. If these are set, you can enable CORBA support (ENABLE\_CORBA) within CMake.

Alternatively, when you use the configure wrapper, you can specify:

```
../configure --enable-corba
```

## 2.5.3. Application Development with TAO

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 -I/path/to/ACE_wrappers/TAO/orbsvcs`

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 -`

ITAO\_PortableServer -ITAO\_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

To quickly test an OrocOS application, you can download the examples from the webpage on Component template [<http://www.orocos.org/ocl/source>], which contains a suitable CMake environment for building components or RTT Examples [<http://www.orocos.org/rtt/source>] which contains a variety of demo programs.

If you built the RTT yourself, you can issue a **make check** in the build directory, which will test the RTT against your current target.

### 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?

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 do not use this function, it is possible that some (OS dependent) Orocos functionality will not work.

### 3.3. Building components and applications

You can quick-start build components using the Orocos Component Template package which you can download from the OCL download page [<http://www.orocos.org/ocl/source>], which uses CMake. If you do not wish to use CMake, you can use the example below to write your own Makefiles.

#### Example 1. A Makefile for an Orocos Application or Component

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

```
OROPATH=/usr/local

all: myprogram mycomponent.so

# Build a purely RTT application for gnulinux.
# Use the 'OCL' settings below if you use the TaskBrowser or other OCL functionality.
#
CXXFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config
orocos-rtt-gnulinux --cflags`
LDFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config
orocos-rtt-gnulinux --libs`

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

# Building dynamic loadable components requires the OCL to be installed as well:
#
CXXFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config
orocos-ocl-gnulinux --cflags`
LDFLAGS=`PKG_CONFIG_PATH=${OROPATH}/lib/pkgconfig pkg-config
orocos-ocl-gnulinux --libs`

mycomonent.so: mycomponent.cpp
    g++ mycomponent.cpp ${CXXFLAGS} ${LDFLAGS} -fPIC -shared
-DOCL_DLL_EXPORT -o mycomponent.so
```

Where your replace *gnulinux* with the target for which you wish to compile. If you use parts of the OCL, use the flags from *orocos-ocl-gnulinux*.

We strongly recommend reading the Deployment Component [<http://www.orocos.org/ocl/deployment>] manual for building and loading Orocos components into an application.

These flags must be extended with compile and link options for your particular application.



### 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, “Detailed Configuration using ‘CMake’” for your target if your application has compilation or link errors ( for example when using LXRT ).

## 3.4. Header Files Overview

**Table 2. Header Files**

Header	Summary
rtt/*.hpp	The ‘Real-Time Toolkit’ directory contains the headers which describe the public API.
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 by the RTT.
rtt/dev/*.hpp	C++ Headers for accessing hardware interfaces.
rtt/corba/*.hpp	C++ Headers for CORBA support.
rtt/scripting/*.hpp	C++ Headers for real-time scripting. Do not include these directly as they are mainly for internal use.
rtt/marsh/*.hpp	C++ Headers for XML configuration and converting data to text and vice versa.
rtt/dlib/*.hpp	C++ Headers for the experimental Distribution Library which allows embedded systems to use some RTT primitives over a network. This directory does not contain such a library but only interface headers.
rtt/impl/*.hpp	C++ Headers for internal use.

## 4. Cross Compiling Orococos

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

Run plain "cmake" or "ccmake" with the following options:

```
CC=cross-gcc CXX=cross-g++ LD=cross-ld cmake ..  
-DCROSS_COMPILE=cross-
```

and substitute the 'cross-' prefix with your target triplet, for example with 'powerpc-linux-gnu-'. This works roughly when running on Linux stations, but is not the official 'CMake' approach.

For having native cross compilation support, you must upgrade to CMake 2.6.0 (not released yet of this writing) or later and follow the instructions on the CMake Cross Compiling page [[http://www.cmake.org/Wiki/CMake\\_Cross\\_Compiling](http://www.cmake.org/Wiki/CMake_Cross_Compiling)].