The OROCOS Real-Time Toolkit Installation Guide

Real-Time Toolkit Version 1.12.1

Revision History

Revision 1.0.0	27 Oct 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.		
Revision 1.4.2	22 Apr 2008	ps
Improved/fixed Debian/Ubuntu package install instructions.		
Revision 1.6.0	02 Aug 2008	kg
Added Mac OS X install instructions.		
Revision 1.10.0	14 Sept 2009	ps
Added pointers to win32 install instructions		
Revision 1.10.1	14 Oct 2009	ps
Clarified instructions for non-standard environments, cleanups.		

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 Orocos build environment

Big Fat Warning

We're gradually moving the contents of the installation manual into the wiki. Check out the The RTT installation wiki [http://www.orocos.org/wiki/rtt/installation] for completeness.

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]), Mac OS X (apple.com [http://www.apple.com/macosx/]) and native Windows using Microsoft Visual Studio. So, you can first write your software as a normal Linux/Mac OS X program, using the framework for testing and debugging purposes in plain userspace (Linux/Mac OS X) and recompile later to a real-time target or MS Windows.

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**
CMake	2.6.0 (all platforms)	See resources on cmake.org [http://www.cmake.org/cmake/resources/software.html] for pre-compiled packages in case your distribution does not support this version
Boost C++ Main Library	1.33.0 (1.36.0 or newer with MS Visual Studio)	Boost.org [http://www.boost.org] Version 1.33.0 has a very efficient (time/space) lock-free smart pointer implementation which is used by Orocos. 1.36.0 has boost::intrusive which we require on Windows with MSVS.
Boost C++ Test Library	1.33.0 (During build only)	Boost.org [http://www.boost.org] test library ('unit_test_framework') is required if you build the RTT from source and BUILD_TESTING=ON (default). The RTT libraries don't depend on this library, it is only used for building our unit tests.
Boost C++ Thread Library	1.33.0 (Mac OS-X only)	Boost.org [http://www.boost.org] thread library is required on Mac OS-X.
GNU gcc / g++ Compilers	3.3.0 (Linux/Cygwin/Mac OS X)	gcc.gnu.org [http://gcc.gnu.org] Orocos builds

Program / Library	***Minimum* Version**	**Description**
		with the GCC 4.x series as well.
MSVS Compilers	2005	One can download the MS VisualStudio 2008 Express edition for free.
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.
ACE & TAO	TAO 1.3 (Optional)	ACE & TAO website [http://www.cs.wustl.edu/~schmidt/] When you start your components in a networked environment, TAO can be used to set up communication between components. CORBA is used as a 'background' transport and is hidden for normal users.
Omniorb	4 (Optional)	Omniorb website [http://omniorb.sourceforge.net/] Omniorb is more robust and faster than TAO, but has less features. CORBA is used as a 'background' transport and is hidden for normal users.

All these packages are provided by most Linux distributions. In Mac OS X, you can install them easily using fink [http://www.finkproject.org] or macports [http://www.macports.org/]. Take also a look on the Orocos.org RTT download [http://www.orocos.org/rtt/source] page for the latest information.

1.2. Basic Real-Time Toolkit Installation on Windows-like systems

We documented this on the on-line wiki for the various flavours/options one has on the MS Windows platform: RTT on MS Windows [http://www.orocos.org/wiki/rtt/rtt-ms-windows]

1.3. Basic Real-Time Toolkit Installation on Unix-like systems

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

The tool you will need is **cmake** Most linux distros have a cmake package, and so do fink/macports in OS X. 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 [http://www.cmake.org].

Next, download the orocos-rtt-1.12.1-src.tar.bz2 package from the Orocos webpage and extract it using :

```
tar -xvjf orocos-rtt-1.12.1-src.tar.bz2
```

This section provides quick installation instructions if you want to install the RTT on a *standard* GNU/Linux system. Please check out Section 2, “Detailed Configuration using 'CMake'” for installation on other OSes and/or if you want to change the default configuration settings.

```
mkdir orocos-rtt-1.12.1/build
cd orocos-rtt-1.12.1/build
cmake .. -DOROCOS_TARGET=<target> [-DCMAKE_INSTALL_PREFIX=/usr/
local] [-DLINUX_SOURCE_DIR=/usr/src/linux]
make
make install
```

Where

- OROCOS_TARGET: <target> is one of 'gnulinux', 'lxrt', 'xenomai', 'macosx', 'win32'. When none is specified, 'gnulinux' is used.
- CMAKE_INSTALL_PREFIX: specifies where to install the RTT.
- LINUX_SOURCE_DIR: 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.

Note

See Section 2, “Detailed Configuration using 'CMake'” for specifying non standard include and library paths to search for dependencies.

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 dochtml** (both in 'build') commands build API documentation and PDF/HTML documentation in the build/doc directory.

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 CMAKE_INSTALL_PREFIX option :

```
cmake .. -DCMAKE_INSTALL_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'

If you have some of the Orocos dependencies installed in non-standard locations, you have to specify this using cmake variables *before* running the cmake configuration. Specify header locations using the CMAKE_INCLUDE_PATH variable (e.g. using bash and fink in Mac OS X, the boost library headers are installed in /sw/include, so you would specify

```
export CMAKE_INCLUDE_PATH=/sw/include;/boost/include
```

For libraries in not default locations, use the

```
export CMAKE_LIBRARY_PATH=/sw/libs;/boost/libs
```

variable. For more information, see cmake useful variables [http://www.cmake.org/Wiki/CMake_Useful_Variables#Environment_Variables] link.

Important

In order to avoid setting these global exports repeatedly, the RTT build system reads a file in which you can specify your build environment. This file is the orocos-rtt.cmake file, which you obtain by making a copy from orocos-rtt-1.12.1/orocos-rtt.default.cmake into the same directory. The advantage is that this file lives in the rtt top source directory, such that it can be re-used across builds. *Using this file is recommended!*

2.1. Real-Time Toolkit Build Configuration

The RTT can be configured depending on your 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-1.12.1/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.

2.1.1. RTT with CORBA plugin

In order to enable CORBA, a valid installation of TAO or OMNIORB 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 and builds and installs an additional library and headers.

Alternatively, you can re-run cmake:

```
cmake .. -DENABLE_CORBA=ON
```

See Section 2.6, “Configuring for CORBA” for full configuration details when using the CORBA transport.

2.1.2. 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).

Warning

The Embedded flavour is not compatible with the Orocos Component Library (OCL) and should only be enabled for specific setups and only by users that understand the consequences of this flag.

Alternatively, when you can re-run cmake:

```
cmake .. -DOS_EMBEDDED=ON
```

2.2. Configuring the target Operating System

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

- gnulinux
- macosx
- xenomai
- lxrt
- win32

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.

2.3. Setting Build Compiler Flags

You can set the compiler flags using the CMAKE_BUILD_TYPE option. You may edit this field to contain:

- Release
- Debug
- RelWithDebInfo
- MinSizeRel
- None

In case you choose None, you must set the CMAKE_C_FLAGS, CMAKE_CXX_FLAGS manually. Consult the CMake manuals for all details.

2.4. 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, 3.5 and 3.6. The last version, RTAI 3.6, is recommended for RTAI users. You can obtain it from the RTAI home page [http://www.rtai.org]. Read The README.* files in the rtai directory for detailed build instructions, as these depend on the RTAI version.

2.4.1. 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'.

2.4.2. 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**.

2.4.3. 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.

2.5. 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'.

2.5.1. 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**.

2.5.2. 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.

2.6. Configuring for CORBA

In case your application benefits from remote access over a network, the RTT can be used with 'The Ace Orb' (*TAO*) or OMNIORB-4. The RTT was tested with TAO 1.3.x, 1.4.x, 1.5x and 1.6.x and OMNIORB 4.1.x. There are two major TAO development lines. One line is prepared by OCI (Object Computing Inc.) [http://www.ociweb.com] and the other by the DOC group [http://www.dre.vanderbilt.edu/]. You can find the latest OCI TAO version on OCI's TAO website [http://www.theaceorb.com]. The DOC group's TAO version can be found on the Real-time CORBA with TAO (The ACE ORB) website [http://www.cs.wustl.edu/~schmidt/TAO.html]. Debian and Ubuntu users use the latter version when they install from .deb packages.

If you need commercial support for any TAO release or seek expert advice on which TAO version or development line to use, consult the commercial support website [http://www.cs.wustl.edu/~schmidt/commercial-support.html].

2.6.1. TAO installation (Optional)

Important

Debian or Ubuntu users can skip this step and just do **sudo aptitude install libtao-orbsvcs-dev tao-idl gperf-ace tao-naming** . Orocos software will automatically detect the installed TAO software.

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.

Orocos requires the ACE, TAO and TAO-orbsvcs libraries and header files to be installed on your workstation. If you used manual installation, *the ACE_ROOT and TAO_ROOT variables must be set.*

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.

2.6.2. Configuring the RTT for TAO or OMNIORB

Orocos RTT defaults to TAO. If you want to use the OMNIORB implementation, run from your build directory:

```
cmake .. -DENABLE_CORBA=ON -DCORBA_IMPLEMENTATION=OMNIORB
```

To specify TAO explicitly (or change back) use:

```
cmake .. -DENABLE_CORBA=ON -DCORBA_IMPLEMENTATION=TAO
```

The RTT will first try to detect your location of ACE and TAO using the ACE_ROOT and TAO_ROOT variables and if these are not set, using the standard include paths. If TAO or OMNIORB is found you can enable CORBA support (ENABLE_CORBA) within CMake.

2.6.3. 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_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

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

```
cmake .. -DENABLE_TESTS=ON
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 you 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 Orocos

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

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 tripplet, 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 or later and follow the instructions on the CMake Cross Compiling page [http://www.cmake.org/Wiki/CMake_Cross_Compiling].