## Installation and Configuration

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Prerequisites</td>
<td>5</td>
</tr>
<tr>
<td>3 Experimental setup</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Installation on Ubuntu 14.04</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Installation on Ubuntu 16.04</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Test your installation</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Orocos tutorial</td>
<td>20</td>
</tr>
<tr>
<td>3.5 Getting started with rtt_lwr 2.0</td>
<td>23</td>
</tr>
<tr>
<td>3.6 MoveIt! with rtt_ros_control_embedded</td>
<td>25</td>
</tr>
<tr>
<td>3.7 Create your first OROCOS controller</td>
<td>26</td>
</tr>
<tr>
<td>3.8 Make you life easier with Kdevelop</td>
<td>29</td>
</tr>
<tr>
<td>3.9 <strong>Update the packages</strong></td>
<td>33</td>
</tr>
<tr>
<td>3.10 Build the documentation</td>
<td>34</td>
</tr>
<tr>
<td>3.11 KRL Tool</td>
<td>34</td>
</tr>
<tr>
<td>3.12 RTT/ROS/KDL Tools</td>
<td>34</td>
</tr>
<tr>
<td>3.13 Bi-Compilation gnulinux/Xenomai</td>
<td>38</td>
</tr>
<tr>
<td>3.14 Compile faster with Ccache and Distcc</td>
<td>41</td>
</tr>
<tr>
<td>3.15 Gazebo Synchronization using Conman</td>
<td>43</td>
</tr>
<tr>
<td>3.16 Gazebo Synchronization using FBSched</td>
<td>43</td>
</tr>
<tr>
<td>3.17 OROCOS with CORBA and MQUEUE</td>
<td>44</td>
</tr>
<tr>
<td>3.18 Xenomai 2.6.5 on Ubuntu 14.04/16.04</td>
<td>44</td>
</tr>
<tr>
<td>3.19 (Beta) Xenomai 3.0.5 on Ubuntu 14.04/16.04</td>
<td>49</td>
</tr>
<tr>
<td>3.20 RTnet setup on Xenomai</td>
<td>55</td>
</tr>
<tr>
<td>3.21 OROCOS RTT on Xenomai</td>
<td>58</td>
</tr>
</tbody>
</table>
rtt_lwr is a set of components for controlling the Kuka LWR and IIWA at 1Khz. It relies on OROCOS for the real-time part, but also interfaces with ROS so we can use Rviz, MoveIt, ros-control etc.

It has been designed so researchers/Phd Students/Engineers at ISIR can develop generic controllers for light weight robots and seemlessly switch between simulation/real hardware without the need to recompile their code.


Prerequisites

• Relatively powerful Ubuntu 16.04/14.04 PC
• Knowledge about ROS
• Notions about OROCOS

Experimental setup

Installation on Ubuntu 14.04

ROS Indigo ++


**Required tools**

```bash
sudo sh -c "echo 'deb http://packages.ros.org/ros/ubuntu $(lsb_release -cs) main' > /etc/apt/sources.list.d/ros-latest.list"
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt update
sudo apt install python-rosdep python-catkin-tools ros-indigo-catkin python-wstool python-vcstool
```

**Fix Locales**

```bash
sudo locale-gen en_US #warnings might occur
sudo locale-gen en_US.UTF-8
sudo nano /etc/environment
# put these lines
LANGUAGE=en_US
LC_ALL=en_US
# Reboot !
```

If you type `perl` you should not see any warnings.
ROS Indigo Desktop

```
# ROS Desktop (NOT DESKTOP-FULL)
sudo apt install ros-indigo-desktop
```

**Warning:** Do not install `desktop-full` (desktop + gazebo 2.2) as we’ll use Gazebo 7.

After Install

```
# Load The environment
source /opt/ros/indigo/setup.bash
# Update ROSdep (to get dependencies automatically)
sudo rosdep init
rosdep update
```

MoveIt! (via debians)

```
# MoveIt!
sudo apt install ros-indigo-moveit
```

MoveIt! (from source)

If you need bleeding-edge features, compile MoveIt! from source:

```
mkdir -p ~/isir/moveit_ws/src
cd ~/isir/moveit_ws/src
# Get all the packages
wstool init
wstool merge https://raw.githubusercontent.com/ros-planning/moveit_docs/indigo-devel/-moveit.rosinstall
wstool update -j2
cd ~/isir/moveit_ws/
# Install dependencies
source /opt/ros/indigo/setup.bash
rosdep install --from-paths ~/isir/moveit_ws/src --ignore-src --rosdistro indigo -y -r
# Configure the workspace
catkin config --init --install --extend /opt/ros/indigo --cmake-args -DCMAKE_BUILD_TYPE=Release
# Build
catkin build
```

OROCOS 2.9 + rtt_ros_integration 2.9 (from source)

If you already completed these instructions, and you are upgrading from orocos 2.8:

- If you installed orocos 2.8 from the debians, you need to remove them `sudo apt remote
  ros-kinetic-orocos-toolchain ros-kinetic-rtt-*`.
- If you installed orocos 2.8 from source, they can live side by side in a different workspace, but always check `catkin config` on your lwr_ws to make sure which workspace you are extending.
Additionally, please make sure that these repos (if you have them) are in the right branches (with fixes for rtt):

```bash
roscd rtt_dot_service && git remote set-url origin https://github.com/kuka-isir/rtt_
˓→dot_service.git && git pull
roscd fbsched && git remote set-url origin https://github.com/kuka-isir/fbsched.git &&
˓→git pull
roscd conman && git remote set-url origin https://github.com/kuka-isir/conman.git &&
˓→git pull
```

### OROCOS toolchain 2.9

```bash
mkdir -p ~/isir/orocos-2.9_ws/src
cd ~/isir/orocos-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_
˓→utils/config/orocos_toolchain-2.9.rosinstall
wstool update -j2
# Get the latest updates (OPTIONAL)
cd orocos_toolchain
git submodule foreach git checkout toolchain-2.9
git submodule foreach git pull
# Configure the workspace
cd ~/isir/orocos-2.9_ws/
# Install dependencies
source /opt/ros/indigo/setup.bash
rosdep install --from-paths ~/isir/orocos-2.9_ws/src --ignore-src --rosdistro indigo -
˓→y -r
catkin config --init --install --extend /opt/ros/indigo/ --cmake-args -DCMAKE_BUILD_
˓→TYPE=Release
# Build
catkin build
```

### rtt_ros_integration 2.9

```bash
mkdir -p ~/isir/rtt_ros-2.9_ws/src
cd ~/isir/rtt_ros-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://github.com/kuka-isir/rtt_lwr/raw/rtt_lwr-2.0/lwr_utils/config/
˓→rtt_ros_integration-2.9.rosinstall
wstool update -j2
# Configure the workspace
cd ~/isir/rtt_ros-2.9_ws/
# Install dependencies
source ~/isir/orocos-2.9_ws/install/setup.bash
rosdep install -q --from-paths ~/isir/rtt_ros-2.9_ws/src --ignore-src --rosdistro indigo -
˓→y -r
catkin config --init --install --extend ~/isir/orocos-2.9_ws/install --cmake-args -
˓→DCMAKE_BUILD_TYPE=Release
# Build (this can take a while)
catkin build
```
Gazebo 7


Note: If you already have gazebo 2.2 installed, please remove it:

```
sudo apt remove gazebo libgazebo-dev ros-indigo-gazebo- *
```

```
# Gazebo 7
curl -ssL http://get.gazebosim.org | sh
# The ros packages
sudo apt install ros-indigo-gazebo7- *
```

Note: Don’t forget to put source source /usr/share/gazebo/setup.sh in your ~/isir/.bashrc or you won't have access to the gazebo plugins (Simulated cameras, lasers, etc).

ROS Control

This allows you to use MoveIt! or just the ros_control capabilities in an orocos environment. Let’s install everything:

```
sudo apt install ros-indigo-ros-control* ros-indigo-control* 
```

RTT LWR packages

```
mkdir -p ~/isir/lwr_ws/src/
cd ~/isir/lwr_ws/src
# Get all the packages
wstool init
# Get rtt_lwr ’base’
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_˓→utils/config/rtt_lwr.rosinstall
# Get the extra packages
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_˓→utils/config/rtt_lwr-extras.rosinstall

# Download
wstool update -j2
```

Note: If you want to install and test cart_opt_ctrl: wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr-utils/config/rtt_lwr-full.rosinstall

Install dependencies

```
source ~/isir/rtt_ros-2.9_ws/install/setup.bash
rosdep install --from-paths ~/isir/lwr_ws/src --ignore-src --rosdistro indigo -y -r 
```
Note: On *indigo*, rosdep will try to install **gazebo 2**, but will fail as we already installed **gazebo 7**. So you can ignore this error if you are running indigo. On ROS kinetic, it will install gazebo7 automatically.

```bash
eventing command [sudo -H apt-get install gazebo2]
Reading package lists... Done
Building dependency tree
Reading state information... Done
Some packages could not be installed. This may mean that you have requested an impossible situation or if you are using the unstable distribution that some required packages have not yet been created or been moved out of Incoming.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
  gazebo2 : Depends: libsdformat-dev (>= 1.4.11-1osrl1) but it is not going to be installed
           Depends: libsdformat-dev (< 2.0.0) but it is not going to be installed
ERROR: the following rosdeps failed to install
  apt: command [sudo -H apt-get install gazebo2] failed
  apt: Failed to detect successful installation of [gazebo2]
```

**Configure the workspace**

```bash
cd ~/isir/lwr_ws
catkin config --init --extend ~/isir/rtt_ros-2.9_ws/install --cmake-args -DCMAKE_BUILD_TYPE=Release
```

**Build the workspace**

Let’s build the entire workspace:

```bash
catkin build --workspace ~/isir/lwr_ws
```
Once it’s done, load the workspace:

```
source ~/isir/lwr_ws/devel/setup.bash
```

**Tip:** Put it in your bashrc:

```
echo 'source ~/isir/lwr_ws/devel/setup.bash' >> ~/.bashrc
```

Now we can **test the installation**.

### Installation on Ubuntu 16.04

**ROS Kinetic ++**


**Required tools**

```
sudo sh -c "echo 'deb http://packages.ros.org/ros/ubuntu $(lsb_release -cs) main' > /etc/apt/sources.list.d/ros-latest.list"
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt update
sudo apt install python-rosdep python-catkin-tools ros-kinetic-catkin python-wstool
```

**Fix Locales**
sudo locale-gen en_US #warnings might occur
sudo locale-gen en_US.UTF-8
sudo nano /etc/environment
# put theses lines
LANGUAGE=en_US
LC_ALL=en_US
# Reboot !

If you type `perl` you should not see any warnings.

**ROS Kinetic Desktop**

```bash
# ROS Desktop (NOT DESKTOP-FULL)
sudo apt install ros-kinetic-desktop
```

**After Install**

```bash
# Load The environment
source /opt/ros/kinetic/setup.bash
# Update ROSdep (to get dependencies automatically)
sudo rosdep init
rosdep update

# MoveIt! (via debians)

```bash
# MoveIt!
sudo apt install ros-kinetic-moveit
```

**OROCOS 2.9 + rtt_ros_integration 2.9 (from source)**

**OROCOS toolchain 2.9**

```bash
mkdir -p ~/isir/orocos-2.9_ws/src
cd ~/isir/orocos-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_utils/config/orocos_toolchain-2.9.rosinstall
wstool update -j2
# Get the latest updates (OPTIONAL)
cd orocos_toolchain
git submodule foreach git checkout toolchain-2.9
git submodule foreach git pull
# Configure the workspace
cd ~/isir/orocos-2.9_ws/
# Install dependencies
source /opt/ros/kinetic/setup.bash
rosdep install --from-paths ~/isir/orocos-2.9_ws/src --ignore-src --rosdistro kinetic
--y --r
catkin config --init --install --extend /opt/ros/kinetic/ --cmake-args -DCMAKE_BUILD_TYPE=Release
```
# Build

```
catkin build
```

## rtt_ros_integration 2.9

```
mkdir -p ~/isir/rtt_ros-2.9_ws/src
cd ~/isir/rtt_ros-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://github.com/kuka-isir/rtt_lwr/raw/rtt_lwr-2.0/lwr_utils/config/
˓→rtt_ros_integration-2.9.rosinstall
wstool update -j2
# Configure the workspace
cd ~/isir/rtt_ros-2.9_ws/
# Install dependencies
source ~/isir/orocos-2.9_ws/install/setup.bash
rosdep install --from-paths ~/isir/rtt_ros-2.9_ws/src --ignore-src --roodistro,
˓→kinetic -y -r
# Configure the workspace
source ~/isir/orocos-2.9_ws/install/setup.bash
rosdep install --from-paths ~/isir/rtt_ros-2.9_ws/src --ignore-src --roodistro,
˓→kinetic -y -r
catkin config --init --install --extend ~/isir/orocos-2.9_ws/install --cmake-args -
˓→DCMAKE_BUILD_TYPE=Release
# Build (this can take a while)
catkin build
```

### Gazebo 8


```
# Gazebo 8
curl -ssL http://get.gazebosim.org | sh
# The ros packages
sudo apt install ros-kinetic-gazebo8-
```

**Note:** Don’t forget to put source `source /usr/share/gazebo/setup.sh` in your `~/isir/.bashrc` or you won’t have access to the gazebo plugins (Simulated cameras, lasers, etc).

### ROS Control

This allows you to use MoveIt! or just the ros_control capabilities in an orocos environnement. Let’s install everything:

```
sudo apt install ros-kinetic-ros-control* ros-kinetic-control*
```

### RTT LWR packages

```
mkdir -p ~/isir/lwr_ws/src/
cd ~/isir/lwr_ws/src
# Get all the packages
wstool init
# Get rtt_lwr 'base'
```
Cart Opt Ctrl

Experimental optimisation based controller:

```bash
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_˓→utils/config/rtt_lwr-full.rosinstall
wstool update
```

Install dependencies

```bash
# If you compiled rtt_ros from sources
source ~/isir/rtt_ros-2.9_ws/install/setup.bash
# Use rosdep tool
rosdep install --from-paths ~/isir/lwr_ws/src --ignore-src --rosdistro kinetic -y -r
```

Note: Gazebo 7 is shipped by default with kinetic, so rosdep will try to install it and fail. You can ignore this issue safely as you now have Gazebo 8 installed.

```bash
# dpkg-query: no packages found matching gazebo7
# ERROR: the following rosdeps failed to install
#   apt: command [sudo -H apt-get install -y gazebo7] failed
#   apt: Failed to detect successful installation of [gazebo7]
```

Configure the workspace

If building rtt_ros from source:

```bash
cd ~/isir/lwr_ws
catkin config --init --extend ~/isir/rtt_ros-2.9_ws/install --cmake-args -DCMAKE_CXX_˓→FLAGS=-std=c++11 -DCMAKE_BUILD_TYPE=Release
```

Build the workspace

Let’s build the entire workspace:

```bash
catkin build --workspace ~/isir/lwr_ws
```
Once it’s done, load the workspace:

```bash
source ~/isir/lwr_ws/devel/setup.bash
```

**Tip:** Put it in your bashrc:

```
echo 'source ~/isir/lwr_ws/devel/setup.bash' >> ~/.bashrc
```

Now we can **test the installation**.

## Test your installation

We’re gonna test a few components to make sure everything is working correctly.

- The **catkin build** during install command has no errors
- Gazebo starts normally
- `lwr_utils` starts normally
- Gazebo inside orocos (embedded) starts normally

**Important:** The Gazebo server in the final setup is launched inside the `gazebo` component (that you can see if you type `ls`). You **will not** be able to launch gazebo separately, it has to be instantiated inside the orocos deployer via this method. This component is provided by the `rtt_gazebo_embedded` package.

## Making sure everything is built

Every one of these commands should provide no errors (all green).
Gazebo

Gazebo needs to get some models at the first launch, so in a terminal type:

```
gzserver --verbose
```

Then `ctrl+C` to close it and type:

```
gazebo
```

Gazebo-ROS

Gazebo with ROS plugin

Start the roscore: `roscore`
Close any instance of gazebo running, and then launch gazebo with the ros plugins:

gazebo -s libgazebo_ros_paths_plugin.so -s libgazebo_ros_api_plugin.so --verbose

Upload the robot’s URDF

roslaunch lwr_description lwr_upload.launch

# you can also pass load_ati_sensor:=true load_handle:=true load_base:=true

Note: If the ROS API is loaded correctly, this command should exit immediately

Spawn to robot into Gazebo

roslaunch lwr_utils spawn_robot.launch robot_name:=lwr_sim

Note: If the model has been correctly uploaded, this command should also exit immediately

Gazebo inside the OROCOS Deployer
**Important:** Close all previous nodes, roscore, deployers, windows etc, and start the main deployer with gazebo.

Now gazebo is launched inside the orocos deployer!

```bash
roslaunch lwr_utils run.launch sim:=true
# this launches gazebo inside the orocos deployer
```

**Note:** You can see the robot in the gazebo gui because the model is spawned into gazebo via the main launch file. Still, it has no interface to send commands, what we’ll do in the next step.

Now type:

```bash
# Load the lwr interface in the deployer
loadRobot(getRobotName(), isSim(), true)
# Let it load, then print the components :
ls
```

You should see all the components running [R]:

![Robot running in Gazebo GUI](image-url)
Orocos tutorial

Orocos Basics

First, make sure you have the lwr workspace loaded

```
source ~/isir/lwr_ws/devel/setup.bash
```

Note: Put this source file in your .bashrc:

```
echo `source ~/isir/lwr_ws/devel/setup.bash` >> ~/.bashrc
```

Official documentation about the Orocos RealTime Toolkit (RTT) can be found here:


The [orocos] Toolchain allows setup, distribution and the building of real-time software components. It is sometimes referred to as 'middleware' because it sits between the application and the Operating System. It takes care of the real-time communication and execution of software components.

All components are ‘deployed’ using a single executable called the ‘deployer’. The main deployer has the ability to load component, connect them to exchange data, start, stop, change their rate etc. To launch it just type in a terminal:

```
deployer
```

Example of a simple 2 components deployment:

```
# Let's load a built-in orocos component
loadComponent("hello","OCL::HelloWorld") # calls the constructor
# If you want to see its ports, properties, attributes :
ls hello
# You can see :
# Data Flow Ports:
# Out(U) string the_results =>
# In(U) string the_buffer_port <= ( use 'the_buffer_port.read(sample)'
# to read a sample from this port)
# It means it has 1 input port and 1 output port
#
# We'll build another component of the same type
loadComponent("hello2","OCL::HelloWorld")
```
# Let's connect their interface
# It's a bi-lateral connection that allow hello1 to connect with hello2's
# ports, attributes etc, and also hello2 to get data from hello
connectPeers("hello","hello2")

# Connect ports
connect("hello.the_buffer_port","hello2.the_results",ConnPolicy())
# The last argument ConnPolicy() is a structure that contains the way
# to send data from one component to another. Default is "DATA"

# Let's run everything

# Setting the activity of
# "hello"
# to a period of 0.1 seconds (10Hz)
# with the thread priority of 10 (0..99)
# It's a standard linux thread
setActivity("hello",0.1,10,ORO_SCHED_OTHER)
setActivity("hello2",0.2,25,ORO_SCHED_OTHER)

# call configureHook()
hello.configure() # registered as an 'operation' called 'configure'
# call updateHook()
hello.start()

hello2.configure()
hello2.start()

# Note that parenthesis are not required for void arguments

# Let's see the data :

hello.the_results.last
# It will show
# "Hello World !"

---

**Tip:** Open a deployer and copy/paste the lines one by one to test.

For further documentation, please refer to the Orocos Builder's Manual.

**Orocos - ROS bridge**

All the magic is done by rtt_ros_integration https://github.com/orocos/rtt_ros_integration. Basically every ROS function that you might be used to call in regular rosnode has been wrapped for orocos to be Real-Time Safe.

Most used features:
• Transform the deployer into a ROS node (rtt_rosnode)
• Connect an Orocos port to a ROS topic (rtt_roscomm)
• Connect an Orocos operation to a ROS service (rtt_roscomm)
• Map Orocos Parameters with the ROS parameter server (rtt_rosparam)
• Get the clock from ros (rtt_rosclock)

Custom Orocos Components with Catkin

Now let’s build our own Orocos Component (Very simple one with no ports, operation nor properties):

```cpp
#include <rtt/RTT.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/Component.hpp>
#include <rtt/Logger.hpp>

class MyComponent : public RTT::TaskContext
{
    // Constructor
    // That’s the name you’re gonna pass as first argument of ”loadComponent”
    public:
    MyComponent(const std::string& name):
    RTT::TaskContext(name)
    {
        RTT::log(RTT::Info) << "Constructing ! " << RTT::endlog();
    }

    // The function called when writing my_component.configure()
    bool configureHook()
    {
        RTT::log(RTT::Info) << "Configuring ! " << RTT::endlog();
    }

    // The function called (periodically or not) when calling my_component.start()
    void updateHook()
    {
        RTT::log(RTT::Info) << "Updating ! " << RTT::endlog();
    }
};
ORO_CREATE_COMPONENT(MyComponent) //Let Orocos know how to build this component
```

The CmakeLists.txt can look like this:

```cpp
cmake_minimum_required(VERSION 2.8.3)
project(my_component)

find_package(catkin REQUIRED COMPONENTS
    # This will automatically import all Orocos components in package.xml,
    # and put them in ${USE_OROCOS_LIBRARIES}
    rtt_ros
cmake_modules
)

include_directories(
    #include
    ${USE_OROCOS_INCLUDE_DIRS}
)
```

Chapter 3. Experimental setup
Then you can just call `cd my_component; mkdir build; cd build; cmake .. && make`. This will generate in the build directory what you can expect from a ROS package: a `devel/` directory containing all the targets (here “my_component”) and a `setup.bash`.

**Note:** Using a catkin workspace makes life much easier: you can put all your packages in `src/`, build them all at once, and you’ll have the `setup.bash` at `my_ws/devel/setup.bash`.

Now if you source `devel/setup.bash` and then call `deployer`, Orocos will know `MyComponent` in its environment:

```python
displayComponentTypes() # You will see MyComponent!
import("rtt_ros")
ros.import("my_component")
loadComponent("my_component","MyComponent")
my_component.configure()
my_component.start()
```

Using `rtt_ros_integration` you can also call:

```python
import("rtt_rospack")
ros.find("my_component")
```


Orocos/ROS documentation for building components easily with catkin: [https://github.com/orocos/rtt_ros_integration](https://github.com/orocos/rtt_ros_integration)

---

### Getting started with rtt_lwr 2.0

#### The main launch file

In `lwr_utils`, you’ll find the main roslaunch to deploy the components:

```bash
roslaunch lwr_utils run.launch sim:=true launch_gazebo:=true
```
This uploads the robot description (tools accessible via `load_*:=true` arguments), the robot_state_publisher (the joint_state_publisher is done by an orocos component called rtt_state_publisher), the service to spawn the robot on gazebo and a few parameters to get the robot name, namespace, tf_prefix etc.

By default, this passes the `utils.ops` script that contains a bunch of useful functions to load the robot and a few components that comes with rtt_lwr. This script is located at `$(rospack find lwr_utils)/scripts/utils.ops`. You can change the script loaded by `run.launch` via the `ops_script:=` argument.

Later on, you’re gonna pass your own customized script as argument:

```
touch my_script.ops && nano my_script.ops and run.launch ops_script:=/path/to/my_script.ops
```

```
import("rtt_rospack")
runScript(ros.find("lwr_utils") + "/scripts/utils.ops")

# loadRobot(getRobotName(),isSim(),true)
# ....
# Your own functions !
```

### Writing your own deployment script (ops file)

A typical sequence for deploying components would be:

```
# Load rospack to find packages in the ros workspace
import("rtt_rospack")
# Load the utility script into the deployer
runScript(ros.find("lwr_utils")+"/scripts/utils.ops")
# Load the robot
```

---

**Chapter 3. Experimental setup**
loadRobot(getRobotName(), isSim(), true)
# Load the state publisher for rviz visualization
loadStatePublisher(true)

# Then you can load your component, connect it etc.

**Note:** Instead of creating everything by hand, please follow the Controller Tutorial and generate a sample project.

Available global functions:

```bash
curl --silent https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_utils/scripts/utils.ops | grep global
```

- `global string getRobotName()`
- `global string getRobotNs()`
- `global string getTfPrefix()`
- `global bool isSim()`
- `global bool setRobotInitialJointConfiguration(double q0, double q1, double q2, double q3, double q4, double q5, double q6)`
- `global string loadKRLTool(bool start_component)`
- `global void setJointTorqueControlMode()`
- `global void setCartesianImpedanceControlMode()`
- `global void setJointImpedanceControlMode()`
- `global bool importRequiredPackages()`
- `global bool connectPortsFromList(string controller_name, string robot_name, strings_ports_list, ConnPolicy cp)`
- `global bool connectStandardPorts(string controller_name, string robot_name, ConnPolicy cp)`
- `global bool connectLWRPorts(string controller_name, string robot_name, ConnPolicy cp)`
- `global bool connectAllPorts(string controller_name, string robot_name, ConnPolicy cp)`
- `global string loadStatePublisher(bool start_component)`
- `global string loadKRLTool(bool start_component)`
- `global string loadConman()`
- `global bool addComponentToStateEstimation(string component_name)`
- `global bool addRobotToConman(string component_name)`
- `global bool addControllerToConman(string component_name)`
- `global string loadFBSched()`
- `global bool addControllerToFBSched(string component_name)`
- `global void generateGraph()`
- `global string loadJointTrajectoryGeneratorKDL(bool start_component)`
- `global string loadROSControl(bool start_component)`
- `global string getAtiFTSensorDataPort()`
- `global bool connectToAtiFTSensorPort(string comp_name, string port_name, ConnPolicy cp)`
- `global string loadAtiFTSensor(bool start_component)`
- `global string loadRobot(string robot_name, bool is_sim, bool start_component)`

**MoveIt! with rtt_ros_control_embedded**

As the other packages, we’ll use the run.launch from lwr_utils that we duplicated in the lwr_moveit_config package for custom arguments.

```bash
roslaunch lwr_moveit_config run.launch sim:=true
```

Then you can list available ros_control controllers:
rtt_lwr Documentation, Release 3.0.0

```
rosrun controller_manager controller_manager list
```

**Note:** These commands launch the following RTT Components:

- gazebo
- lwr_sim
- rtt_ros_control_embedded
  - controller_manager
  - hardware interface

Then you have access to the full ros_control interface as a normal ROS module, so you can create your own ros_controllers. [http://wiki.ros.org/ros_control](http://wiki.ros.org/ros_control).

---

**Create your first OROCOS controller**

**Using lwr_create_pkg**

Using the lwr_project_creator utility, we can generate a (very) simple controller for our robot.

**Generate the controller**

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
lwr_create_pkg my_controller -C MyController
cd ~/catkin_ws
catkin config --init --extend ~/isir/lwr_ws/devel
```
Build the controller

```bash
cd ~/catkin_ws
catkin build
# add the controller to the ros package path
source ~/catkin_ws/devel/setup.bash

Launch it:

roslaunch my_controller run.launch sim:=true
```
General note on controllers

This generated controller is “kuka-lwr-free” and as a general rule you should avoid to include kuka-lwr stuff in your code.

You should just see the robot as an *Effort interface*, i.e., you should only listen to the current state (q, qdot) and publish torques.

**Recommended** ports are:

<table>
<thead>
<tr>
<th>Robot input / Controller output:</th>
<th>JointTorqueCommand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(JointPositionCommand)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot Output / Controller input:</th>
<th>JointPosition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JointVelocity</td>
</tr>
<tr>
<td></td>
<td>(JointTorque)</td>
</tr>
</tbody>
</table>

The complete list of LWR ports:

```
curl --silent https://raw.githubusercontent.com/kuka-isir/lwr_hardware/master/lwr_fri/src/FRIComponent.cpp | grep -oP 'addPort\(".*"\)'
```

- CartesianImpedanceCommand
- CartesianWrenchCommand
- CartesianPositionCommand
- JointImpedanceCommand
- JointTorqueCommand → To Controller
- toKRL → KRL Tool
- KRL_CMD → KRL Tool
- fromKRL → KRL Tool
- CartesianWrench
The controller uses `rtt_ros_kdl_tools::ChainUtils` to create an “arm” object. This arm loads the robot_description from the ROS parameter server (you can use the provided launch file that helps you start everything), then create a few KDL chain, solvers etc to compute forward kinematics, jacobians etc.

Functions available can be found [here](#).

Inverse Kinematics is included in ChainUtils via Trac IK.

**Make you life easier with Kdevelop**

Kdevelop is a nice IDE that supports cmake natively. To install it, just type `sudo apt install kdevelop`, then in a terminal, type `kdevelop`.

**Tip:** To enable sourcing the bashrc from the Ubuntu toolbar, `sudo nano /usr/share/applications/kde4/kdevelop.desktop` and replace `Exec=kdevelop %u` by `Exec=bash -i -c kdevelop %u`.

**Import a catkin/CMake project**

Click on Project --&gt; Open/Import Project...
Select the `CMakeLists.txt` inside your project.

Select the `CMakeLists.txt` inside your project.

Select **you project** as the root directory.
Correct the Build Directory if necessary, and if you’ve already built with `catkin build`, you should see `Using an already created build directory`.

**Warning:** Make sure the Build Directory is set to `/path/to/catkin_ws/build/my_project`.

Click on finish and you’re done import your project.

---

3.8. Make you life easier with Kdevelop
Build your project

On the vertical left panel, click on Projects and you’ll see the list of your currently opened projects. Yours should appear here after a few seconds.

You can check also in the Build Sequence that your project appears. To build, click on build :)
Note: Clicking on build is equivalent to calling:

```
cd /path/to/catkin_ws
catkin build my_super_project
```

### Update the packages

We are periodically doing updates on the code (new tools, bug fixes etc), so keeping it up-to-date can be very useful.

To update all the packages we’re going to use the `vcs-tools` utility ([https://github.com/dirk-thomas/vcstool](https://github.com/dirk-thomas/vcstool)).

It should be already installed during the `installation` procedure, otherwise `sudo apt install python-vcstool`.

#### Update OROCOS

```
cd ~/isir/orocos-2.9_ws/src
vcs pull
# update submodules for bleeding-edge updates (OPTIONAL)
vcs-git submodule foreach git checkout toolchain-2.9
vcs-git submodule foreach git pull
```
Update RTT ROS Integration

```bash
cd ~/isir/rtt_ros-2.9/src
vcs pull
```

Update rtt_lwr

```bash
cd ~/isir/lwr_ws/src
vcs pull
vcs-git submodule update
```

Build the documentation

Documentation is built using readthedocs, using itself sphinx, and hosted on https://readthedocs.org/. Everytime a push is made on github, a new build is triggered at http://rtt-lwr.readthedocs.io/.

Syntaxe

It uses reStructuredText instead of default markdown in order to support math equations, notes, warnings etc. More info at https://docs.readthedocs.io/en/latest/getting_started.html

Generate documentation locally

```bash
pip install -U --user pip sphinx sphinx-autobuild recommonmark sphinx_rtd_theme
```

- Go to the docs directory: roscd rtt_lwr/../docs and type make livehtml.
- Open your favorite webbrowser and go to 127.0.0.1/8000 to see the generated site locally.

Update the doc using your favorite text editor, like atom (https://atom.io), and open a pull request on github at https://github.com/kuka-isir/rtt_lwr.

KRL Tool

RTT/ROS/KDL Tools

Available at https://github.com/kuka-isir/rtt_ros_kdl_tools

This set of tools helps you build and use a robot kinematic model via a provided URDF.

Robot Kinematic Model

The ChainUtils class provides tools to compute forward kinematics, external torques etc. It devoices from ChainUtils-Base and adds non-realtime inverse kinematics with trac_ik.
Frames of reference / ROS params

To build the kinematic chain, we’ll use the following ROS params:

- /robot_description (the URDF uploaded)
- /root_link (default/recommended: /link_0)
- /tip_link (example: /ati_link)

All those parameters must live in the relative namespace (http://wiki.ros.org/Names) from your node to build the chain with KDL.

Note: If you use the rtt_lwr tools, like the generated run.launch linked to lwr_utils, the params are automatically sent when you’ll run your controller.

Build your ChainUtils model

In your package.xml:

```xml
<build_depend>rtt_ros_kdl_tools</build_depend>
<run_depend>rtt_ros_kdl_tools</run_depend>
```

In your CMakeLists.txt:

```cmake
# Add the rtt_ros dependency
find_package(catkin REQUIRED COMPONENTS rtt_ros)
# It's gonna import all orocos libraries in the package.xml
```

In your .hpp:

```cpp
#include <rtt_ros_kdl_tools/chain_utils.hpp>
```

In your .cpp:

```cpp
// Construct your model
rtt_ros_kdl_tools::ChainUtils arm;

// Initialise chain from robot_description (via ROS Params)
// It reads /robot_description
// /tip_link
// /root_link
arm.init();
```
You need to tell `ChainUtils` the state of the robot and compute some internal model variables.

// Ex : read from orocos port in an Eigen::VectorXd
port_joint_position_in.read(jnt_pos_in); // Check if RTT::NewData
port_joint_velocity_in.read(jnt_vel_in); // Check if RTT::NewData

// Feed the internal state
arm.setState(jnt_pos_in, jnt_vel_in);
// Update the internal model
arm.updateModel();
Forward kinematics

ChainUtils arm

Joint Positions
q

Forward Kinematics
F(q)

Segment Position w.r.t root link
X

// Ex : read from orocos port in an Eigen::VectorXd
port_joint_position_in.read(jnt_pos_in); // Check if RTT::NewData
port_joint_velocity_in.read(jnt_vel_in); // Check if RTT::NewData

// Feed the internal state
arm.setState(jnt_pos_in, jnt_vel_in);
// Update the internal model
arm.updateModel();

// Ex : get a specific segment position
KDL::Frame & X = arm.getSegmentPosition("link_7");

// Ex : get the 5th segment
KDL::Frame & X = arm.getSegmentPosition(5);

// Get Root Link
std::string root_link = arm.getRootSegmentName();

Inverse kinematics with trac_ik

ChainUtils arm

Tip Link Position
X

Inverse Kinematics
F(X)

Joint Positions
q

// Ex : read from orocos port in an Eigen::VectorXd
port_joint_position_in.read(jnt_pos_in); // Check if RTT::NewData
port_joint_velocity_in.read(jnt_vel_in); // Check if RTT::NewData
// Feed the internal state
arm.setState(jnt_pos_in, jnt_vel_in);
// Update the internal model
arm.updateModel();

// Transform it to a KDL JntArray
KDL::JntArray joint_seed(jnt_pos_in.size());
joint_seed.data = jnt_pos_in;

// Allocate the output of the Inverse Kinematics
KDL::JntArray return_joints(jnt_pos_in.size());

// Create an desired frame
KDL::Frame desired_end_effector_pose(
    KDL::Rotation::RPY(-1.57, 0, 1.57), // Rotation rad
    KDL::Vector(-0.2, -0.3, 0.8)); // Position x,y,z in meters

// Define some tolerances
KDL::Twist tolerances(
    KDL::Vector(0.01, 0.01, 0.01), // Tolerance x,y,z in meters
    KDL::Vector(0.01, 0.01, 0.01)); // Tolerance Rx,Rz,Rz in rad

// Call the inverse function
if (arm.cartesianToJoint(joint_seed,
                        desired_end_effector_pose,
                        return_joints,
                        tolerances))
{
    log(Debug) << "Success ! Result : " << return_joints.data.transpose() << endlog();
}

Warning: trac-ik is not realtime-safe and therefore should not be used in updateHook()

Bi-Compilation gnulinux/Xenomai

Tutorial: You are using a gnulinux computer and would like to try to build for xenomai

1. You can install a xenomai kernel and go all xenomai. Please refer to the Xenomai section <\>.

2. You can build your components to xenomai and test it on another xenomai computer later.

First, get Xenomai libs:

tar xfvj xenomai-2.6.5.tar.bz2
cd xenomai-2.6-5
mkdir install
./configure --prefix=`pwd`/install
make -j`nproc`
make install
# Now Xenomai is installed in ~/xenomai-2.6-5/install

Then clean everything in your orocos workspace:
cd orocos-2.9_ws/
catkin clean -y

Then add new profiles:

```bash
catkin profile add xenomai
catkin profile add gnulinux
```

If you run catkin profile list:

```
[profile] Available profiles:
xenomai (active)
default
gnulinux
```

For Xenomai

Use the xenomai profile: `catkin profile set xenomai`.

Let’s configure the workspace, the important option is `--env-cache` as it will save the current environment in a static file (inside orocos-2.9_ws/.catkin_tools). It is only saving when running catkin build!

```bash
catkin config --extend /opt/ros/indigo 
   --install 
   --env-cache 
   -b build-xenomai 
   -d devel-xenomai 
   -i install-xenomai 
   --cmake-args -DCMAKE_BUILD_TYPE=Release 
   -DENABLE_CORBA=ON -DENABLE_MQ=ON -DCORBA_IMPLEMENTATION=OMNIORB
```

Note that we put the build, devel and install space in a different folder so it won’t collide with gnulinux builds.

Now you can build!

```bash
OROCOS_TARGET=xenomai XENOMAI_ROOT_DIR=~/xenomai-2.6.5/install catkin build
```

**Note:** Now the env var OROCOS_TARGET and XENOMAI_ROOT_DIR are `set/fixed` for all the packages. If you modify them using export, it won’t affect the ones written for the packages during build.

Once it’s done you’ll have:

```
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ ll
total 44
  drwxrwxr-x 15 hoarau hoarau 4096 Jul 27 16:36 ..
  drwxrwxr-x 10 hoarau hoarau 4096 Jul 27 22:50 build-xenomai/
  drwxrwxr-x  3 hoarau hoarau 4096 May 2 10:13 .catkin_tools/
  drwxrwxr-x  5 hoarau hoarau 4096 Jul 27 16:57 devel-xenomai/
  drwxrwxr-x  7 hoarau hoarau 4096 Jul 27 22:50 install-xenomai/
  drwxrwxr-x 11 hoarau hoarau 4096 Jul 27 22:40 logs/
  drwxrwxr-x  4 hoarau hoarau 4096 Jun  7 11:55 src/
```

And inside install-xenomai/lib you’ll have -xenomai libraries:
For gnulinux

Let’s use the default profile: `catkin profile set default` Let’s configure it the standard way:

```
catkin config --extend /opt/ros/indigo
   --env-cache
   -b build
   -d devel
   --install
   -i install
   --cmake-args -DCMAKE_BUILD_TYPE=Release -DENABLE_CORBA=ON
   -DENABLE_MQ=ON -DCORBA_IMPLEMENTATION=OMNIORB
```

And build it:

```
OROCOS_TARGET=gnulinux catkin build
```

We now have two version of the same libs (gnulinux/xenomai):

```
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ ll
 total 44
 drwxrwxr-x 11 hoarau hoarau 4096 Jul 27 22:48 ./
 drwxrwxr-x 15 hoarau hoarau 4096 Jul 27 16:36 ../
 drwxrwxr-x 11 hoarau hoarau 4096 Jul 27 22:40 build/
 drwxrwxr-x 10 hoarau hoarau 4096 Jul 27 22:50 build-xenomai/
 drwxrwxr-x  3 hoarau hoarau 4096 May  2 10:13 .catkin_tools/
 drwxrwxr-x  5 hoarau hoarau 4096 Jul 27 22:40 devel/
 drwxrwxr-x  5 hoarau hoarau 4096 Jul 27 16:57 devel-xenomai/
 drwxrwxr-x  7 hoarau hoarau 4096 Jul 27 22:41 install/
 drwxrwxr-x  7 hoarau hoarau 4096 Jul 27 22:50 install-xenomai/
 drwxrwxr-x 11 hoarau hoarau 4096 Jul 27 22:40 logs/
 drwxrwxr-x  4 hoarau hoarau 4096 Jun  7 11:55 src/
```

Now let’s prove it works:

```
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ catkin profile list
[profile] Available profiles:
xenomai (active)
default
gnulinux

# Let's check the global linux variable
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ echo $OROCOS_TARGET
gnulinux

# The next command will show the env variable written on the package rtt cache
# (enabled via catkin config --env-cache)
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ catkin build --get-env rtt | grep ORO
typeset -x OROCOS_TARGET=xenomai
```

We are on xenomai profile, the global linux OROCOS_TARGET is set to gnulinux, but the env written on each package (here rtt) is xenomai!
You can also try to launch to launch the deployer on non-xenomai kernel:

```bash
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ source install-xenomai/setup.sh
hoarau@waro-G55VW:~/ros_ws/orocos-2.9_ws$ deployer
Xenomai: /dev/rtheap is missing
(chardev, major=10 minor=254)
# It does not work, because you don't have a xenomai kernel
```

Final note: on kuka-rt2 (xenomai kernel) it can support both versions!

### Compile faster with Ccache and Distcc

- **ccache**: [https://ccache.samba.org/](https://ccache.samba.org/)
- **distcc**: [https://github.com/distcc/distcc](https://github.com/distcc/distcc)

**Note:** You need to be connected to a high speed network to actually improve your build time

### Installation on hosts and clients

```bash
sudo apt install distcc ccache distccmon-gnome
```

### Build server configuration

On every build computer, you need to install `distcc` and `ccache` and update the configuration:

```bash
sudo nano /etc/default/distcc
```

Then set the following variables:

```bash
STARTDISTCC="true"
#
# Which networks/hosts should be allowed to connect to the daemon?
# You can list multiple hosts/networks separated by spaces.
# Networks have to be in CIDR notation, e.g. 192.168.1.0/24
# Hosts are represented by a single IP address
#
# ALLOWEDNETS="127.0.0.1"
# This means allow from myself and computers from 192.168.1.0 to 192.168.1.255
ALLOWEDNETS="127.0.0.1 192.168.1.0/24"
```

Restart the distcc daemon and you are all set up! `sudo service distcc restart`.

### Client configuration

```bash
export CCACHE_PREFIX="distcc"
# Use gcc 4.8 on 14.04
export CC="ccache gcc-4.8" CXX="ccache g++-4.8"
# Use gcc 5 on 16.04
```
export CC="ccache gcc-5" CXX="ccache g++-5"
# If you want only distcc use this
#export CC="distcc gcc-4.8" CXX="distcc g++-4.8"
#export CC="distcc gcc-5" CXX="distcc g++-5"

# List here all the known distcc servers/number of threads
export DISTCC_HOSTS='localhost/4 kuka-viz/6 kuka-viz2/6'

# Build your workspace with
catkin build -p$(distcc -j) -j$(distcc -j) --no-jobserver

Tip: Put this alias in your ~/.bashrc: alias cdistcc="catkin build -p$(distcc -j) -j$(distcc -j) --no-jobserver"

Warning: The command in CC and CXX are the commands sent over the network. In order to avoid compiler conflicts, put the compiler version (ex: gcc-4.8)!

Building a catkin workspace with Distcc

If you’ve already built your workspace without distcc, you’ll need to clean it first catkin clean -y. Then, verify you have the variables set correctly (as above): echo $CXX && echo $CC.

Now you can use catkin build -p$(distcc -j) -j$(distcc -j) --no-jobserver to build your workspace.

Tip: You can use distccmon-gnome to visualize the distribution.
Gazebo Synchronization using Conman

Installation

```bash
mkdir -p ~/isir/conman_ws/src
cd ~/isir/conman_ws/src
wstool init
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr-utils/config/conman.rosinstall
wstool update -j2
cd ~/conman_ws/
# Disable tests on 14.04
catkin config --init --cmake-args -DCATKIN_ENABLE_TESTING=0 -DCMAKE_BUILD_TYPE=Release
# Build
catkin build
```

Gazebo Synchronization using FBSched

As an example you can launch `roslaunch rtt_joint_traj_generator_kdl run.launch sim:=true`. Then in the deployer's console:
# Then make sure gazebo follows its executionEngine
gazebo.use_rtt_sync = true

# load the fbs component (default 1Khz)
loadFBSched()

# Set a 1Hz period for the test
fbs.setPeriod(1.0)

# Then for each component
addComponentToFBSched("gazebo")

# This one is not necessary as it is gazebo's slave
addComponentToFBSched("lwr_sim")

# The controller
addComponentToFBSched("lwr_sim_traj_kdl")

# Just for the fun of it
addComponentToFBSched("lwr_sim_state_pub")
addComponentToFBSched("lwr_sim_kri_tool")

# Configure and start FBSched
fbs.configure()
fbs.start()

Note: When “fbs” starts, gazebo catches up with all the world’s callbacks, so it might jump forward in time. If you put this code in you ops, and pause gazebo, then you won’t have any issue.

OROCOS with CORBA and MQUEUE

In order to use the corba interface (connect multiple deployers together), you’ll need to build the orocos_ws and rtt_ros_ws with:

```
catkin config --cmake-args -DCMAKE_BUILD_TYPE=Release -DENABLE_MQ=ON -DENABLE_CORBA=ON -DCORBA_IMPLEMENTATION=OMNIORB
```


Xenomai 2.6.5 on Ubuntu 14.04/16.04

Recommended Hardware

- Intel/AMD Processor i5/i7 (< 2016 is recommended to guarantee full 16.04 support)
- Dedicated Ethernet controller for RTnet, with either e1000e/e1000/r8169 driver (Intel PRO/1000 GT recommended)

Warning: Nvidia/Ati Drivers are NOT supported (creates a lot of interruptions that breaks the real-time constraints). Please consider removing the dedicated graphic card and use the integrated graphics (Intel HD graphics).
Download Xenomai 2.6.5


tar xfvj xenomai-2.6.5.tar.bz2

Download Linux kernel 3.18.20

wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.18.20.tar.gz

tar xfv linux-3.18.20.tar.gz

Note: We chose 3.18.20 because it is the latest kernel compatible with xenomai 2.6.5.

Configuration

Prevent a bug in make-kpkg in 14.04

From https://bugs.launchpad.net/ubuntu/+source/kernel-package/+bug/1308183:

cd linux-3.18.20

# Paste that in the terminal

cat <<EOF > arch/x86/boot/install.sh
#!/bin/sh

cp -a -- "\$2" "\$4/vmlinuz-\$1"

EOF

Prepare the kernel

sudo apt install kernel-package

Patch the Linux kernel with Xenomai ipipe patch


cd linux-3.18.20

./xenomai-2.6.5/scripts/prepare-kernel.sh

Press Enter to use the default options.

Configure the kernel

Now it’s time to configure:

Gui version:

make xconfig

Or without gui:
Some guidelines to configure the Linux kernel:

### Recommended options:

- **General setup**
  - Local version - append to kernel release: `-xenomai-2.6.5`

- **Timers subsystem**
  - High Resolution Timer Support (Enable)

- **Real-time sub-system**
  - Xenomai (Enable)
  - Nucleus (Enable)
    - Pervasive real-time support in user-space (Enable)

- **Power management and ACPI options**
  - Run-time PM core functionality (Disable)
  - ACPI (Advanced Configuration and Power Interface) Support
    - Processor (Disable)
  - CPU Frequency scaling
    - CPU Frequency scaling (Disable)
  - CPU idle
    - CPU idle PM support (Disable)
  - Processor type and features
    - Processor family
      - Core 2/newer Xeon (if cat /proc/cpuinfo | grep family returns 6, set as)
  - Generic otherwise

- **Memory power savings**
  - Intel chipset idle memory power saving driver (Disable)

**Warning:** For OROCOS, we need to increase the amount of resources available for Xenomai tasks, otherwise we might hit the limits quickly as we add multiples components/ports etc. [http://www.orocos.org/forum/orocos/orocos-users/orocos-limits-under-xenomai](http://www.orocos.org/forum/orocos/orocos-users/orocos-limits-under-xenomai)

- **Real-time sub-system**
  - Number of registry slots
    - 4096
  - Size of the system heap
    - 2048 Kb
  - Size of the private stack pool
    - 1024 Kb
  - Size of private semaphores heap
    - 48 Kb
  - Size of global semaphores heap
    - 48 Kb

Save the config and close the gui.

**Compile the kernel (make debians)**

Now it’s time to compile.
CONCURRENCY_LEVEL=$(nproc) make-kpkg --rootcmd fakeroott --initrd kernel_image kernel_headers

Take a coffee and come back in 20min.

Compile faster with distcc

If you have distcc servers setup and a fast network, you can speed up drastically the building speed.

MAKEFLAGS="CC=distcc" BUILD_TIME="/usr/bin/time" CONCURRENCY_LEVEL=$(distcc -j) make-kpkg --rootcmd fakeroott --initrd kernel_image kernel_headers

Install the kernel

cd ..
sudo dpkg -i linux-headers-3.18.20-xenomai-2.6.5_3.18.20-xenomai-2.6.5-10.00.Custom_amd64.deb linux-image-3.18.20-xenomai-2.6.5_3.18.20-xenomai-2.6.5-10.00.Custom_amd64.deb

Allow non-root users

sudo addgroup xenomai --gid 1234
sudo addgroup root xenomai
sudo usermod -a -G xenomai $USER

Tip: If the addgroup command fails (ex: GID xenomai is already in use), change it to a different random value, and report it in the next section.

Configure GRUB

Edit the grub config :

sudo nano /etc/default/grub

GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 3.18.20-xenomai-2.6.5"
#GRUB_DEFAULT=saved
#GRUB_SAVEDefault=true
#GRUB_HIDDEN_TIMEOUT=0
#GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=5
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash xeno_nucleus.xenomai_gid=1234"
GRUB_CMDLINE_LINUX=""

Note: Please note the xenomai group (here 1234) should match what you set above (allow non-root users).
**Tip:** the `noapic` option might be added if the screen goes black at startup and you can’t boot.

If you have an Intel HD Graphics integrated GPU (any type):

```bash
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash i915.enable_rc6=0 i915.powersave=0 noapic
 → xeno_nucleus.xenomai_gid=1234 xenomai.allowed_group=1234"
# This removes powersavings from the graphics, that creates disturbing interruptions.
```

If you have an Intel Skylake (2015 processors), you need to add `nosmap` to fix the latency hang (https://xenomai.org/pipermail/xenomai/2016-October/036787.html):

```bash
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash i915.enable_rc6=0 i915.powersave=0 xeno
 → nucleus.xenomai_gid=1234 nosmap"
```

Update GRUB and reboot

```
sudo update-grub
dudo reboot
```

### Install Xenomai libraries

```
cd xenomai-2.6.5/
./configure
make -j$(nproc)
sudo make install
```

Update your `bashrc`

```
echo '#### Xenomai
export XENOMAI_ROOT_DIR=/usr/xenomai
export XENOMAI_PATH=/usr/xenomai
export PATH=$PATH:$XENOMAI_PATH/bin
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:$XENOMAI_PATH/lib/pkgconfig
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$XENOMAI_PATH/lib
export OROCOS_TARGET=xenomai
' >> ~/.bashrc
```

### Test your installation

```
xeno latency
```

This loop will allow you to monitor a xenomai latency. Here’s the output for a i7 4Ghz:

```
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 99)
RTT|----lat min|----lat avg|----lat max|--overrun|--msw|--lat best|--lat worst
RTD| 0.174| 0.464| 1.780| 0| 0| 0.174| 1.780
RTD| 0.088| 0.464| 1.357| 0| 0| 0.088| 1.780
RTD| 0.336| 0.464| 1.822| 0| 0| 0.088| 1.822
```
Tip: To get pertinent results, you need to stress your system while running the latency test. The latency has to be stable even if the system is under load.

Tip: To get pertinent results, you need to stress your system while running the latency test. The latency has to be stable even if the system is under load.

```
sudo apt install stress
  # Using stress
  stress -v -c 8 -i 10 -d 8
```

**Negative latency issues**

You need to be in root sudo -s, then you can set values to the latency calibration variable in nanoseconds:

```
$ echo 0 > /proc/xenomai/latency
  # Now run the latency test

  # If the minimum latency value is positive,
  # then get the lowest value from the latency test (ex: 0.088 us)
  # and write it to the calibration file (here you have to write 88 ns):
  $ echo my_super_value_in_ns > /proc/xenomai/latency
```

Source: https://xenomai.org/pipermail/xenomai/2007-May/009063.html

**(Beta) Xenomai 3.0.5 on Ubuntu 14.04/16.04**

These instructions demonstrates how to build a Cobalt Core for Xenomai 3.0.5.

Official documentation is available at https://xenomai.org/installing-xenomai-3-x/

**Recommended Hardware**

- **Intel/AMD Processor i5/i7** (< 2016 is recommended to guarantee full 16.04 support)
- **Dedicated Ethernet controller for RTnet**, with either e1000e/e1000/r8169 driver (Intel PRO/1000 GT recommended)

**Warning:** Nvidia/Ati Drivers are NOT supported (creates a lot of interruptions that breaks the real-time constraints). Please consider removing the dedicated graphic card and use the integrated graphics (Intel HD graphics).

**Get the linux kernel**

```
wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.9.38.tar.gz
  tar xf linux-4.9.38.tar.gz
```

3.19. (Beta) Xenomai 3.0.5 on Ubuntu 14.04/16.04
Note: We chose 4.9.38 because it is the latest kernel compatible with xenomai 3.0.5. You can find the patches at https://xenomai.org/downloads/ipipe/

Get Xenomai 3.0.5

```bash
tar xf xenomai-3.0.5.tar.bz2
```

Apply the Xenomai patch

We assume you are building an `x86_64` kernel (64bits).

```bash
cd linux-4.9.38
../xenomai-3.0.5/scripts/prepare-kernel.sh --arch=x86_64 --ipipe=ipipe-core-4.9.38-x86-3.patch
```

Configure the kernel

Take the actual working config:

```bash
yes "" | make oldconfig
```

Gui version:

```bash
make xconfig
```

Or GTK+ version:

```bash
sudo apt install gtk+-2.0 glib-2.0 libglade2-dev
make gconfig
```

Or without gui:

```bash
sudo apt install libncurses5-dev
make menuconfig
```

Some guidelines to configure the linux kernel:

Recommended options:

- General setup
  - Local version - append to kernel release: -xenomai-3.0.5
  - Timers subsystem
  - High Resolution Timer Support (Enable)
- Xenomai/cobalt
  - Sizes and static limits
  - Number of registry slots (512 --> 4096)
  - Size of system heap (Kb) (512 --> 4096)
  - Size of private heap (Kb) (64 --> 256)
  - Size of shared heap (Kb) (64 --> 256)
  - Maximum number of POSIX timers per process (128 --> 512)
--- Drivers
  -- RTnet
    --> RTnet, TCP/IP socket interface (Enable)
    --> Drivers New Intel(R) PRO/1000 PCIe (Enable)
    --> Add-Ons
      --> Real-Time Capturing Support (Enable)
  * Power management and ACPI options
    --> CPU Frequency scaling
      --> CPU Frequency scaling (Disable)
    --> ACPI (Advanced Configuration and Power Interface) Support
      --> Processor (Disable)
    --> CPU Idle
      --> CPU idle PM support (Disable)
  * Processor type and features
    --> Enable maximum number of SMP processors and NUMA nodes (Disable)
    // Ref: http://xenomai.org/pipermail/xenomai/2017-September/037718.html
    --> Processor family
      --> Core 2/newer Xeon (if "cat /proc/cpuinfo | grep family" returns 6, set as Generic otherwise)
    // Xenomai will issue a warning about CONFIG_MIGRATION, disable those in this order
    --> Transparent Hugepage Support (Disable)
    --> Allow for memory compaction (Disable)
    --> Contiguous Memory Allocation (Disable)
    --> Allow for memory compaction
      --> Page Migration (Disable)
  * Device Drivers
    --> Staging drivers
      --> Unisys SPAR driver support
        --> Unisys visorbus driver (Disable)

---

**Warning:** Unlike xenomai 2.x, RTnet has to be built in the kernel. Make sure to choose the drivers correctly.

---

**Build the Real-Time kernel**

```
sudo apt install kernel-package
CONCURRENCY_LEVEL=$(nproc) make-kpkg --rootcmd fakeroot --initrd kernel_image kernel_headers
```

---

**Compile faster with distcc**

If you have distcc servers setup and a fast network, you can speed up drastically the building speed.

```
MAKEFLAGS="-C=/usr/lib/distcc/gcc-5" make-kpkg -j$(distcc -j) --rootcmd fakeroot --initrd kernel_image kernel_headers
```

---

**Note:** Always set gcc version for distcc so that the server can figure out which one to choose (it may have gcc-4.8 by default)

```
cd ..
sudo dpkg -i linux-headers-4.9.38-xenomai-3.0.5_4.9.38-xenomai-3.0.5-10.00.Custom_amd64.deb
```

---

3.19. (Beta) Xenomai 3.0.5 on Ubuntu 14.04/16.04
### Allow non-root users

```bash
sudo addgroup xenomai --gid 1234
sudo addgroup root xenomai
sudo usermod -a -G xenomai $USER
```

**Tip:** If the `addgroup` command fails (ex: GID `xenomai` is already in use), change it to a different random value, and report it in the next section.

### Configure GRUB and reboot

Edit the grub config :

```bash
sudo nano /etc/default/grub
```

```bash
GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 4.9.38-xenomai-3.0.5"
#GRUB_DEFAULT=saved
#GRUB_SAVEDEFAULT=true
# Comment the following lines
#GRUB_HIDDEN_TIMEOUT=0
#GRUB_HIDDEN_TIMEOUT_QUIET=true
GRUB_TIMEOUT=5
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash xenomai.allowed_group=123"
GRUB_CMDLINE_LINUX=""
```

**Note:** Please note the xenomai group (here 1234) should match what you set above (allow non-root users).

**Tip:** noapic option might be added if the screen goes black at startup and you can’t boot.

If you have an Intel HD Graphics integrated GPU (any type) :

```bash
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash i915.enable_rc6=0 i915.enable_dc=0 noapic_˓→xenomai.allowed_group=123"
# This removes powersavings from the graphics, that creates disturbing interruptions.
```

If you have an Intel **Skylake** (2015 processors), you need to add nosmap to fix the latency hang ([https://xenomai.org/pipermail/xenomai/2016-October/036787.html](https://xenomai.org/pipermail/xenomai/2016-October/036787.html)) :

```bash
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash i915.enable_rc6=0 i915.enable_dc=0 xeno_˓→nucleus.xenomai_gid=1234 nosmap"
```

Update GRUB and reboot

```bash
sudo update-grub
sudo reboot
```
Installing Xenomai 3.0.5 User space libraries

First, make sure that you are running the cobalt kernel:

```
uname -a
# Should return Linux waro-rt 4.9.38-xenomai-3.0.5 #2 SMP Wed Sep 20 16:00:12 CEST 2017 x86_64 x86_64 x86_64 GNU/Linux
dmesg | grep Xenomai
# [ 1.417024] [Xenomai] scheduling class idle registered.
# [ 1.417025] [Xenomai] scheduling class rt registered.
# [ 1.417045] [Xenomai] disabling automatic C1E state promotion on Intel processor
# [ 1.417055] [Xenomai] SMI-enabled chipset found, but SMI workaround disabled
# [ 1.417088] I-pipe: head domain Xenomai registered.
# [ 1.417704] [Xenomai] allowing access to group 1234
# [ 1.417726] [Xenomai] Cobalt v3.0.5 (Sisyphus's Boulder) [DEBUG]
```

cd xenomai-3.0.5
./configure --with-pic --with-core=cobalt --enable-smp --disable-tls --enable-dlopen-libs --disable-clock-monotonic-raw
make -j`nproc`
sudo make install

Prevent Gazebo compiling issues

Proper fix : use Xenomai git repo

The issue has been fixed with commits 5ec8be8 and 1ed69a6.

```
git clone https://git.xenomai.org/xenomai-3.git
cd xenomai-3
./script/bootstrap
./configure --with-pic --with-core=cobalt --enable-smp --disable-tls --enable-dlopen-libs
make -j`nproc`
sudo make install
```

Hack with v3.0.5

Gazebo won’t compile because of some conflicting macros ( clz() ) present in libtbb and libcobalt. Remove this macro from xenomai to hack-fix it. It is only used in xenomai internals so won’t cause any issue in user-land. use this hack only if you install xenomai from the 3.0.5 package, not the git repo.

```
sudo sed -i 's/clz/__clz/g' /usr/xenomai/include/boilerplate/compiler.h
```

Update your bashrc

```
echo '### Xenomai
export XENOMAI_ROOT_DIR=/usr/xenomai
```
Test your installation

```
xeno latency

This loop will allow you to monitor a xenomai latency. Here’s the output for a i7 4Ghz:
```

```
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT| 00:00:01 (periodic user-mode task, 100 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| 0.174| 0.464| 1.780| 0| 0| 0.174| 1.780
RTD| 0.088| 0.464| 1.357| 0| 0| 0.088| 1.780
RTD| 0.336| 0.464| 1.822| 0| 0| 0.088| 1.822
RTD| 0.342| 0.464| 1.360| 0| 0| 0.088| 1.822
RTD| 0.327| 0.462| 2.297| 0| 0| 0.088| 2.297
RTD| 0.347| 0.463| 1.313| 0| 0| 0.088| 2.297
RTD| 0.314| 0.464| 1.465| 0| 0| 0.088| 2.297
RTD| 0.190| 0.464| 1.311| 0| 0| 0.088| 2.297

Tip: To get pertinent results, you need to stress your system while running the latency test. The latency has to be stable even if the system is under load.

```
sudo apt install stress
# Using stress
stress -v -c 8 -i 10 -d 8
```

Fix negative latency issues

You need to be in root sudo -s, then you can set values to the latency calibration variable in nanoseconds:

```
$ echo 0 > /proc/xenomai/latency
# Now run the latency test

# If the minimum latency value is positive,
# then get the lowest value from the latency test (ex: 0.088 us)
# and write it to the calibration file (here you have to write 88 ns):
$ echo my_super_value_in_ns > /proc/xenomai/latency
```

Source: https://xenomai.org/pipermail/xenomai/2007-May/009063.html
RTnet setup on Xenomai

Official website: http://www.rtnet.org/index.html

RTnet allows you to send and receive data with very strict constraints, in a real-time environment (RTAI, Xenomai). It only works with a very limited set of ethernet cards (RTnet includes “real-time” re-written drivers): Intel PRO/1000, 82574L, any card with r8169 (xenomai < 2.6.4 only) and others.

First make sure you followed the Xenomai installation instructions and you are running the Xenomai kernel (uname -a).

Recommended hardware

- Intel Pro/1000 GT: e1000e driver ← Recommended
- D-Link DGE-528T: r8169s driver

Check which kernel driver you use

```
lspci -vvv -nn | grep -C 10 Ethernet
```

And check if the rt_ version exists in RTnet’s drivers.

Note: We’re using a custom website that fixes compilation problems for kernel > 3.10 source.

Download

```
sudo apt install git
```

```
git clone https://github.com/kuka-isir/RTnet.git
```

Installation

We’ll need the following options:

- Variant
  --> Xenomai 2.1
- Protocol Stack
  --> TCP Support (for ATI Force Sensor)
- Drivers
  --> The driver you use (New Intel PRO/1000 in our case)
  --> Loopback (optional)
- Add-Ons
  --> Real-Time Capturing Support (optional, for Wireshark debugging)
- Examples
  --> RTnet Application Examples
  --> Enable (optional)
sudo apt install libncurses5-dev

cd RTnet
make menuconfig
    # Configure the options below
    # Then hit ESC 2 times, save the config, and build
make
    # Install in /usr/local/rtnet/ (default location)
sudo make install

Configuration

The configuration file is located by default at /usr/local/rtnet/etc/rtnet.conf. Take a look at this configuration file.

- **RT_DRIVER="rt_e1000e"** The driver we use (we have the Intel PRO/1000 GT)
- **REBIND_RT_NICS="0000:05:00.0 0000:06:00.0"** NIC addresses of the 2 cards we use for RTnet (you can check the NIC address typing `lshw -C network` and looking at “bus info: pci@...”. It is useful to have a fix master/slave config order (card1->robot, card2->Sensor for example).
- **IPADDR="192.168.100.101"** IP of the master (your computer). All the slaves will send/receive to/from master IP.
- **NETMASK="255.255.255.0"** The other slave will have IPs 192.168.100.XXX.
- **RT_LOOPBACK="no"** Not used now. Might be useful to use it somehow.
- **RT_PROTOCOLS="udp packet tcp"** Robot sends via UDP, ATI Sensor via TCP for config, UDP otherwise.
- **RTCAP="yes"** To debug with Wireshark
- **TDMA_CYCLE="450"** and **TDMA_OFFSET="50"** Data from robot/sensor takes about 350us to receive (using Wireshark).

Allow non-root users

To allow commands like `rtnet start` etc to be used without `sudo`, we will use `visudo`. We remove password in certain commands only for people in the xenomai group.

```
sudo visudo
    # then add the following at the end
%xenomai ALL=(root) NOPASSWD:/sbin/insmod
%xenomai ALL=(root) NOPASSWD:/sbin/rmmod
%xenomai ALL=(root) NOPASSWD:/sbin/modprobe
%xenomai ALL=(root) NOPASSWD:/bin/echo
%xenomai ALL=(root) NOPASSWD:/bin/mknod
%xenomai ALL=(root) NOPASSWD:/usr/bin/service
%xenomai ALL=(root) NOPASSWD:/usr/sbin/service
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtcfg
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtifconfig
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtiwconfig
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtnet
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtping
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/rtroute
%xenomai ALL=(root) NOPASSWD:/usr/local/sbin/tdmacfg
```
Test your installation

Using the test script

A launch script can be found here. Adjust the following settings to your needs:

- SLAVES="192.168.100.102 192.168.100.103"
- SLAVES_NAMES="Kuka ATISensor"

Then to use it:

```
./path/to/script/rtnet start
```

Manually

```
cd /usr/local/rtnet/sbin
# Start the rt kernel drivers
sudo ./rtnet start
# Bringup connection
sudo ./rtifconfig rteth0 up 192.168.100.101 netmask 255.255.255.0
# Bringup slaves
sudo ./rtroute solicit 192.168.100.102 dev rteth0
# Ping Slave
sudo ./rtping 192.168.100.102
# Stop everything
sudo ./rtnet stop
```

**Note:** You might have to remove the non-rt kernel driver before rtnet start:

```
sudo rmmod e1000e
sudo ./rtnet start
```

**Note:** You should see `rt_e1000e` as the kernel driver currently used

```
lspci -vvv -nn | grep -C 10 Ethernet
lsmod | grep rt_
```

Use RTnet in C++

The API is the same as regular socket in C, except that the functions start with `rt_*`. To make sure it compiles on every platform, add the following to your headers:

```
#ifndef HAVE_RTNET

// Rename the standard functions
// And use theses ones to be RTnet-compatible when available

#define rt_dev_socket socket
#define rt_dev_setsockopt setsockopt
#define rt_dev_bind bind

#endif
```

3.20. RTnet setup on Xenomai
#define rt_dev_recvfrom recvfrom
#define rt_dev_sendto sendto
#define rt_dev_close close
#define rt_dev_connect connect

#else
// Use RTnet in Xenomai
#include <rtdm/rtdm.h>
#endif

And in your CMakeLists.txt (example):

```cmake
# Add the path to the FindRTnet.cmake folder
# Let's assume you put it in /path/to/project/cmake
list(APPEND CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)

if(${ENV{OROCOS_TARGET} STREQUAL "xenomai")
  find_package(RTnet)
  if(NOT ${RTnet_FOUND})
    message(ERROR "RTnet cannot be used without Xenomai")
  else()
    message(STATUS "RTnet support enabled")
    set_property(TARGET ${TARGET_NAME} APPEND PROPERTY COMPILATION_DEFINITIONS HAVE_RTNET
    -XENOMAI)
  endif()
endif()
```

Note: FindRTnet.cmake can be found here.

---

**OROCOS RTT on Xenomai**

Orocos RTT is fully compatible with Xenomai, but it needs to be built from source.

The only difference in the build process is the need to set `export OROCOS_TARGET=xenomai` before compiling.

**Orocos Toolchain 2.9 on Xenomai**

```bash
# Compile for Xenomai
export OROCOS_TARGET=xenomai

mkdir -p ~/isir/orocos-2.9_ws/src
cd ~/isir/orocos-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://raw.githubusercontent.com/kuka-isir/rtt_lwr/rtt_lwr-2.0/lwr_/utils/config/orocos_toolchain-2.9.rosinstall
wstool update -j2
# Get the latest updates (OPTIONAL)
cd orocos_toolchain
git submodule foreach git checkout toolchain-2.9
git submodule foreach git pull
```
Rtt ROS integration 2.9 on Xenomai

```bash
# Compile for Xenomai
export OROCOS_TARGET=xenomai

mkdir -p ~/isir/rtt_ros-2.9_ws/src
cd ~/isir/rtt_ros-2.9_ws/src
# Get all the packages
wstool init
wstool merge https://github.com/kuka-isir/rtt_lwr/raw/rtt_lwr-2.0/lwr_utils/config/ → rtt_ros_integration-2.9.rosinstall
wstool update -j2

cd ~/isir/rtt_ros-2.9_ws/
# Install dependencies
source ~/isir/orocos-2.9_ws/install/setup.bash
rosdep install --from-paths ~/isir/rtt_ros-2.9_ws/src --ignore-src --rosdistro kinetic → y -r
# Configure the build in Release (recommended), and enable extra transport methods → (optional)
catkin config --init --install --extend ~/isir/orocos-2.9_ws/install --cmake-args -DCMAKE_BUILD_TYPE=Release -DENABLE_MQ=ON -DENABLE_CORBA=ON -DCORBA_IMPLEMENTATION=OMNIORB
# Build
catkin build
```

Tip: Always build for Xenomai on the Xenomai computer: `echo 'export OROCOS_TARGET=xenomai' `>> ~/.bashrc

Note: Libraries built with xenomai will have `-xenomai.so` appended to the library name (ex `libmycontroller-xenomai.so`)