RoNeX Documentation

Release indigo-devel

Shadow Robot Software Team

January 25, 2017

Contents

CHAPTER 1

General RoNeX Setup

1.1 Setting up your computer

1.1.1 Groovy Install

We assume you've installed ROS groovy following those instructions.

- You'll need to install the following packages to be able to build from source.

```
sudo apt-get install python-rosdep python-wstool build-essential
sudo rosdep init
rosdep update
```

- You can now download the rosinstall file - this file contains all the information needed to download the different source of the packages that need to be built. We'll assume the file has been downloaded to `~/Downloads/sr_ronex.rosinstall`.
- Let's create a catkin workspace to download and compile the different packages. We'll use `~/catkin_ws` for this step by step (but it could be anywhere).

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
wstool init
wstool merge ~/Downloads/ronex-groovy.rosinstall
wstool update
rosdep install --from-paths src --ignore-src --rosdistro groovy -y
```

- To load the workspace, you need to source the setup.bash:

```
source ~/catkin_ws/devel/setup.bash
```

- If you want the workspace to be sourced each time you open a terminal, then you can source it in your `~/.bashrc`. To do this run:

```
echo "source ~/catkin_ws/devel/setup.bash" >~/.bashrc
```

- It is now time to build all those packages (this can take a long time)

```
cd ~/catkin_ws
catkin_make_isolated
```

Once this command finishes successfuly, the RoNeX drivers are installed. You can continue reading the wiki to get started with your RoNeX.

The current RoNeX driver is based on ROS. We recommend that you use a computer with Ubuntu LTS 14.04 ("Trusty") installed on it. You will need less than 1GB of additional disk space for all of the ROS components. Follow the instructions here: Installing ROS Indigo to install ROS Indigo on Ubuntu - we recommend the `ros-indigo-desktop-full` package.

1.1.2 Connecting RoNex

RoNex can be connected to your computer via an ethernet port, however if you are lacking spare ports, a USB 2.0 to Gigabit ethernet adapter can be used. We have tested RoNex with the StarTech.com USB 2.0 to Gigabit Ethernet NIC Network Adapter.

1.1.3 Installing the drivers for ROS Indigo

This is the **easiest and recommended way** to install the RoNeX drivers. To install the drivers and their dependencies, simply run:

```
sudo apt-get install ros-indigo-sr-ronex
```

Using ROS is made much easier by setting up the terminal window and its environment. If indigo is the only ROS version on the computer then you should do:

```
sudo echo "source /opt/ros/indigo/setup.bash" >> -/.bashrc
```

Now each new terminal window will have the ROS Indigo environment setup automatically.

If you need to have multiple versions of ROS installed, then using this method would force everyone to use Indigo! If you don't want to do that, then in each new terminal window you should type:

```
source /opt/ros/indigo/setup.bash
```

1.1.4 Installing the drivers for ROS Groovy

Due to some changes to the pr2 code base that were integrated in ROS Indigo only, if you want to use RoNeX with ROS Groovy, you'll have to install it from sources. This is a more time consuming approach. If you need to go down this route, please follow these instructions.

1.1.5 Setting up a Catkin workspace

If you're planning on developing code with RoNeX, chances are you'll be using Catkin to build your programs. Therefore you're going to need a Catkin workspace, which can be set up following these instructions.

Don't forget to source the workspace devel/setup.bash file in your .bashrc file to add the packages in the workspace to the ROS environment.

1.2 Launching the RoNeX driver

To start the driver, you'll need elevated permissions. Run:

```
sudo -s
roslaunch sr_ronex_launch sr_ronex.launch
```

By default the driver is looking for the RoNeX on **eth0**. If you want to start it another interface, you can add the correct ethercat port argument to the launch command:

```
roslaunch sr_ronex_launch sr_ronex.launch ethercat_port:=eth1
```

1.3 General System Overview

1.3.1 Using aliases with your RoNeX

You can give aliases (i.e., pseudonyms) to your RoNeX modules (such as General I/O modules). An alias will be used as a unique identifier for one specific RoNeX module instead of its serial number.

Set an alias for a RoNeX module

Start a roscore in a terminal:

```
roscore
```

Then in another terminal, set the RoNeX module to use the alias `test_ronex`. In the following command, we use `1234` as the serial number - replace it with your own.

```
rosparam set /ronex/mapping/1234 "test_ronex"
```

Now you can start the RoNeX driver, and the ronex_id will reflect the alias above. If the driver is already running, you will need to restart it in order for the changes to take effect. Additionally, when you stop the roscore, the parameters will be erased and you will need to set the parameter again before running the driver next time. If you are planning on using the same alias constantly, you can set this parameter at the start of the sr_ronex.launch file by adding the following line after the first tag:

```
<param name="/ronex/mapping/1234" value="test_ronex" />
```

This page describes some of the key parameters, services and topics in the RoNeX system. If you're unfamiliar with these concepts, you can read more about the ROS system architecture here. If you're new to ROS you may also find it helpful to run through some of the ROS Wiki Tutorials.

This page mostly describes the GIO module which is a very general module. More specialised modules, like the SPI module, will present additional interfaces or different types of interfaces to the user, so you should review their specific documentation. The serial number of the sample RoNeX GIO module used here was 12. On other systems the number 12 will be replaced with the serial number corresponding to the connected GIO module.

1.3.2 Topics

ROS Topics provide a fast, flexible way to transfer data between nodes.

Topics can be published or subscribed to from the command line or within a program. For more information see the General Examples and Tutorials.

- `/diagnostics` : RoNeX publishes diagnostics messages containing information on connected modules and communication status to this topic.

1.3.3 Parameters

The parameter server acts as a database of information to be shared between various nodes running within your ROS system.

Any of these parameters can be read using the rosparam get command at any time.

- `/robot_description`: This parameter will contain a copy of your robot's URDF file, to be used by amongst other things, joint controllers and visualisation tools.
- `/ronex/devices/0/path`: The path to the first detected RoNeX device, all parameters, topics and services related to this device will start with this path.
- `/ronex/devices/0/product_id`: This parameter tells you the product_id number corresponding to the type of module detected.
- `/ronex/devices/0/product_name`: This parameter tells you the name corresponding to the type of module detected.
- `/ronex/devices/0/ronex_id`: This parameter will contain the id of the RoNeX module, which will be the module alias if you have set one (as described here, otherwise it will match the serial number.)
- `/ronex/devices/0/serial`: The pre-programmed serial number of the RoNeX module, which makes it possible to distinguish between different modules of the same type on the bus.

1.4 Interacting with RoNeX

There are four main methods to interact with RoNeX, this section will give a brief introduction to these methods.

1.4.1 Command Line

We can access these interfaces using ROS's command line tools, which are convenient for getting a quick overview of what is going on in the system.

1.4.2 Python and C++

Once you're familiar with the basic operation of the RoNeX from the command line, we can take a step to automate the interactions by writing them into a program. This page features examples using both Python and C++. While coding in Python is sometimes simpler, it is mainly down to user preference when selecting which language to use.

1.4.3 ROS GUI

There are a number of tools included in the rqt_gui packages, that allow us to interact with RoNeX through the standard ROS GUI, without having to go through typing out the terminal commands repeatedly. These can be handy for plotting received data, or repeatedly sending commands. To start the gui, run:

```
rosrun rqt_gui rqt_gui
```

1.5 Generic RoNeX Tutorials

1.5.1 Accessing Module parameters [Command Line]

The ROS parameter server stores details of each RoNeX module such as serial number, module type, module alias etc. These can be read using the rosparam commands as follows.

To list all of the entries currently in the parameter server:

```
rosparam list
```

You should see a number of entries under /ronex/devices for each module you have connected. You can view all of the details for one module (the first on the bus in this case) like so:

```
rosparam get /ronex/devices/0
```

Which will return something along the lines of:

```
{path: /ronex/general_io/12, product_id: '33554433', product_name: general_io, ronex_id: '12',
  serial: '12'}
```

You can also read the input mode for each of the channels along with some other info for each module, but beware that as these are dynamic parameters, changing them with the rosparam set command will not work. You should instead use the dynparam interfaces as explained in the Changing GIO Configuration tutorials.

1.5.2 Accessing Module parameters [Python]

The example demonstrates how to access RoNeX modules listed in the parameter server. For each module, the parameter server stores parameters such as its product_id, product_name, ronex_id, path, and serial. Note that we assume here that RoNeX consists of a Bridge (IN) module, and one or multiple General I/O module(s), although all modules will display similar attributes.

The code

First change directories to your **sr_ronex_examples** package.

```
roscd sr_ronex_examples/
```

Python file **sr_ronex_parse_parameter_server.py** is located inside the **src** directory.

```
#!/usr/bin/env python

import rospy

class SrRonexParseParamExample(object):

    def __init__(self):
        self.find_general_io_modules()

    def find_general_io_modules(self):
        """
        Find the General I/O modules present on the system.
        """
        while True:
            try:
                rospy.get_param("/ronex/devices/0/ronex_id")
```

```
                break
            except:
                rospy.loginfo("Waiting for the General I/O module to be loaded properly.")
                sleep(0.1)

        devices = rospy.get_param("/ronex/devices")
        for ronex_param_id in devices:
            rospy.loginfo( "*** General I/O Module %s ***",  ronex_param_id );
            rospy.loginfo( "product_id   = %s", devices[ronex_param_id]["product_id"] );
            rospy.loginfo( "product_name = %s", devices[ronex_param_id]["product_name"] );
            rospy.loginfo( "ronex_id     = %s", devices[ronex_param_id]["ronex_id"] );
            rospy.loginfo( "path         = %s", devices[ronex_param_id]["path"]);
            rospy.loginfo( "serial       = %s", devices[ronex_param_id]["serial"] );

if __name__ == "__main__":

    rospy.init_node("sr_ronex_parse_parameter_server")
    SrRonexParseParamExample()
```

The Code Explained

```
if __name__ == "__main__":

    rospy.init_node("sr_ronex_parse_parameter_server")

    SrRonexParseParamExample()
```

`rospy.init_node(NAME)` is very important as it tells rospy the name of your node – until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name **sr_ronex_parse_parameter_server**.

The next line `SrRonexParseParamExample()` launches the demo by creating an object of class **SrRonexParseParamExample**.

```
while True:
          try:
              rospy.get_param("/ronex/devices/0/ronex_id")
              break
          except:
              rospy.loginfo("Waiting for the General I/O module to be loaded properly.")
              sleep(0.1)
```

Loop until at least one General I/O module has been properly loaded.

```
devices = rospy.get_param("/ronex/devices")
for ronex_param_id in devices:
    rospy.loginfo( "*** General I/O Module %s ***",  ronex_param_id );
    rospy.loginfo( "product_id   = %s", devices[ronex_param_id]["product_id"] );
    rospy.loginfo( "product_name = %s", devices[ronex_param_id]["product_name"] );
    rospy.loginfo( "ronex_id     = %s", devices[ronex_param_id]["ronex_id"] );
    rospy.loginfo( "path         = %s", devices[ronex_param_id]["path"]);
    rospy.loginfo( "serial       = %s", devices[ronex_param_id]["serial"] );
```

Retrieve information about all loaded General I/O modules stored in a dictionary (with `ronex_param_id` as its keyword). By iterating through all values of `ronex_param_id`, we can retrieve the information about each General I/O module's product_id, product_name, ronex_id, path, and serial. Note that if `ronex_id` (its type is string) has not been set to an alias name, its value is equal to the value of `serial`.

Running the code

First make sure that the RoNeX driver is running (see Launching the RoNeX driver).

Once this is done we can run our Python script:

```
rosrun sr_ronex_examples sr_ronex_parse_parameter_server.py
```

You will see something similar to:

```
[INFO] [WallTime: 1380010917.786130] *** General I/O Module 0 ***
[INFO] [WallTime: 1380010917.786482] product_id = 33554433
[INFO] [WallTime: 1380010917.786716] product_name = general_io
[INFO] [WallTime: 1380010917.786938] ronex_id = 2
[INFO] [WallTime: 1380010917.787193] path = /ronex/general_io/2
[INFO] [WallTime: 1380010917.787444] serial = 12
```

1.5.3 Accessing Module parameters [C++]

The example demonstrates how to access RoNeX modules listed in the parameter server. For each module, the parameter server stores parameters such as its product_id, product_name, ronex_id, path, and serial. Note that we assume here that RoNeX consists of a Bridge (IN) module, and one or multiple General I/O module(s), although all modules will display similar attributes.

The code

First change directories to your **sr_ronex_examples** package.

```
roscd sr_ronex_examples
```

C++ file **sr_ronex_parse_parameter_server.cpp** is located inside the **src** directory.

```
#include <string>
#include <ros/ros.h>
#include <ros/console.h>
#include <boost/lexical_cast.hpp>

#include "sr_ronex_utilities/sr_ronex_utilities.hpp"

class SrRonexParseParamExample
{
public:
  SrRonexParseParamExample()
  {
    find_general_io_modules_();
  }

  ~SrRonexParseParamExample() {}

private:

  void find_general_io_modules_(void)
  {
    ros::Rate loop_rate(10);
    std::string param;
    while ( ros::param::get("/ronex/devices/0/ronex_id", param ) == false )
    {
```

```
      ROS_INFO_STREAM( "Waiting for General I/O module to be loaded properly.\n" );
      loop_rate.sleep();
    }

    std::string empty_ronex_id("");
    int next_ronex_parameter_id = ronex::get_ronex_param_id(empty_ronex_id);

    for (int ronex_parameter_id = 0;
         ronex_parameter_id < next_ronex_parameter_id;
         ronex_parameter_id++)
    {

      std::string product_id;
      std::string product_id_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("
      ros::param::get( product_id_key, product_id );

      std::string product_name;
      std::string product_name_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string
      ros::param::get( product_name_key, product_name );

      std::string path;
      std::string path_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("path")
      ros::param::get( path_key, path );

      std::string ronex_id;
      std::string ronex_id_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("ro
      ros::param::get( ronex_id_key, ronex_id );

      std::string serial;
      std::string serial_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("seri
      ros::param::get( serial_key, serial );

      ROS_INFO_STREAM( "*** General I/O module " << ronex_parameter_id << " ***" );
      ROS_INFO_STREAM( "product_id   = " << product_id );
      ROS_INFO_STREAM( "product_name = " << product_name );
      ROS_INFO_STREAM( "ronex_id     = " << ronex_id );
      ROS_INFO_STREAM( "path         = " << path );
      ROS_INFO_STREAM( "serial       = " << serial );
    }
  }

  std::string to_string_(int d)
  {
    return boost::lexical_cast<std::string>(d);
  }
};

int main(int argc, char **argv)
{

  ros::init(argc, argv, "sr_ronex_parse_parameter_server");
  ros::NodeHandle n;
  SrRonexParseParamExample example;

  return 0;
}
```

The Code Explained

```
ros::init(argc, argv, "sr_ronex_parse_parameter_server");
```

Initialize ROS. This allows ROS to do name remapping through the command line – not important here. This is also where we specify the name of our node. Node names must be unique in a running system. The name used here must be a base name (i.e., it cannot have a / in it).

```
ros::NodeHandle n;
```

Create a handle to this process' node. The first NodeHandle created will actually do the initialization of the node, and the last one destructed will cleanup any resources the node was using.

```
ros::Rate loop_rate(10);
std::string param;
while (ros::param::get("/ronex/devices/0/ronex_id", param ) == false)
{
  ROS_INFO( "Waiting for General I/O module to be loaded properly." );
  loop_rate.sleep();
}
```

Loop until at least one General I/O module has been properly loaded.

```
std::string empty_ronex_id("");
int next_ronex_parameter_id = ronex::get_ronex_param_id(empty_ronex_id);
```

This C++ version is more complicated than the Python version, because parameters are NOT stored in a dictionary as in Python.

Index `ronex_parameter_id` starts from 0. When an empty string is given to `ronex::get_ronex_param_id` as the input argument, it returns the next available `ronex_parameter_id`.

```
for (int ronex_parameter_id = 0; ronex_parameter_id <
     next_ronex_parameter_id; ronex_parameter_id++)
{
  std::string product_id; std::string product_id_key = ronex::get_ronex_devices_string( ronex_paramet
                                                                                        std::string("
  ros::param::get( product_id_key, product_id );
  std::string product_name;
  std::string product_name_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("pr
  ros::param::get( product_name_key, product_name );

  std::string path;
  std::string path_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("path") );
  ros::param::get( path_key, path );

  std::string ronex_id;
  std::string ronex_id_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("ronex_
  ros::param::get( ronex_id_key, ronex_id );

  std::string serial;
  std::string serial_key = ronex::get_ronex_devices_string( ronex_parameter_id, std::string("serial")
  ros::param::get( serial_key, serial );

  ROS_INFO_STREAM( "*** General I/O module " << ronex_parameter_id << " ***" );
  ROS_INFO_STREAM( "product_id   = " << product_id );
  ROS_INFO_STREAM( "product_name = " << product_name );
  ROS_INFO_STREAM( "ronex_id     = " << ronex_id );
  ROS_INFO_STREAM( "path         = " << path );
```

```
  ROS_INFO_STREAM( "serial       = " << serial );
}
```

We retrieve the values of all parameters (i.e., product_id, product_name, ronex_id, path, and serial) related to the General I/O module, and output the data to console.

Note that if ronex_id (its type is string) has not been set to an alias name, its value is equal to the value of serial. And serial is an integer that starts from 1.

Running the code

Make sure that a roscore is up and running:

```
roscore
```

If you're running this code from your own workspace, you'll first need to build it using Catkin, if you're not sure how to do this you can follow the instructions here.

Next sure that a roscore and the RoNeX driver are running (see Launching the RoNeX driver).

Once this is done we can run our C++ program:

```
rosrun sr_ronex_examples sr_ronex_parse_parameter_server
```

You will see something similar to:

```
[INFO] [1380018712.243856548]: *** General I/O module 0 ***
[INFO] [1380018712.243969375]: product_id = 33554433
[INFO] [1380018712.244016969]: product_name = general_io
[INFO] [1380018712.244051559]: ronex_id = 2
[INFO] [1380018712.244087449]: path = /ronex/general_io/2
[INFO] [1380018712.244124994]: serial = 2
```

1.5.4 Create a package to interact with RoNeX

In this tutorial, we will create a package called **my_first_package** (inside workspace **my_first_ws**) to interact with RoNeX. Make sure that you read the following tutorials first.

1. Creating a workspace for catkin
2. Creating a catkin package

This tutorial assumes that you have installed catkin and sourced your environment.

```
source /opt/ros/hydro/setup.bash
```

Creating a catkin workspace

Let's create a catkin workspace:

```
mkdir -p ~/my_first_ws/src
cd ~/my_first_ws/src
catkin_init_workspace
Creating symlink "~/my_first_ws/src/CMakeLists.txt" pointing to "/opt/ros/hydro/share/catkin/cmake/to
```

Even though the workspace is empty (there are no packages in the **src** folder, just a single CMakeLists.txt link) you can still "build" the workspace:

```
cd ~/my_first_ws/
catkin_make
```

Before continuing source your new setup.*sh file:

```
source devel/setup.bash
```

Creating a catkin Package

Change directories to the **src** folder (i.e., THE SOURCE SPACE) in your workspace:

```
cd ~/my_first_ws/src
```

Now use the **catkin_create_pkg** script to create a new package called **my_first_package** which depends on std_msgs, sr_ronex_msgs, sr_ronex_utilities, dynamic_reconfigure, roscpp, and rospy.

```
catkin_create_pkg my_first_package std_msgs sr_ronex_msgs sr_ronex_utilities dynamic_reconfigure rosc
```

This will create a **my_first_package** folder which contains a package.xml and a CMakeLists.txt, which have been partially filled out with the information you gave catkin_create_pkg.

Writing the Source Code

For this tutorial, we simply copy **sr_ronex_flash_LED_with_PWM.cpp** from the **sr_ronex_examples** package to the **src** folder, and rename it to **sr_ronex_flash_LED_with_PWM_2.cpp**. You can find the source code in the PWM tutorial.

```
ls ~/my_first_ws/src/my_first_package/src
sr_ronex_flash_LED_with_PWM_2.cpp
```

Edit the file (i.e., **sr_ronex_flash_LED_with_PWM_2.cpp**) and change the node name we give to `ros::init`.

```
int main(int argc, char **argv)
{
  // Initialize ROS with a unique node name.
  ros::init(argc, argv, "sr_ronex_flash_LED_with_PWM_2");

  // Create a handle to this process' node.
  ros::NodeHandle n;
```

In **sr_ronex_flash_LED_with_PWM_2.cpp**, we use the `advertise()` function to tell ROS that we want to publish on a given topic name. Note that this version of advertise is a templated convenience function.

```
void flash_LED( ros::NodeHandle& n, const std::string& topic )
{
  ros::Publisher pub = n.advertise<sr_ronex_msgs::PWM>( topic, 1000 );

  // .........
}
```

Of course, we have to tell the compiler where it can find the definition of `sr_ronex_msgs::PWM`.

```
#include "sr_ronex_msgs/PWM.h"
```

If you want to know more about `sr_ronex_msgs::PWM`, you can use **rosmsg show**.

```
rosmsg show sr_ronex_msgs/PWM
uint16 pwm_period
uint16 pwm_on_time_0
uint16 pwm_on_time_1
```

Customizing Your Package

```
ls ~/my_first_ws/src/my_first_package
CMakeLists.txt  include  package.xml  src
```

Customizing the package.xml

Follow these instructions to customize the package.xml.

Customizing the CMakeLists.txt

Declare things to be passed to dependent projects. You must invoke catkin_package() before adding any targets (libraries and executables). The reason is because catkin_package() will change the location where the targets are built.

```
catkin_package(
  CATKIN_DEPENDS dynamic_reconfigure roscpp rospy sr_ronex_msgs sr_ronex_utilities std_msgs
)
```

Add the following lines to the end of **CMakeLists.txt**.

```
add_executable(sr_ronex_flash_LED_with_PWM_2 ${PROJECT_SOURCE_DIR}/src/sr_ronex_flash_LED_with_PWM_2
add_dependencies(sr_ronex_flash_LED_with_PWM_2 sr_ronex_msgs_gencpp ${PROJECT_NAME}_gencfg)
target_link_libraries(sr_ronex_flash_LED_with_PWM_2 ${catkin_LIBRARIES})
```

Running the code

Change directories to your RoNeX workspace, and compile the code.

```
cd ~/my_first_ws
catkin_make
```

Now use **rosrun** to run the code (after starting **roscore** in another terminal).

```
source ~/my_first_ws/devel/setup.bash
rosrun my_first_package sr_ronex_flash_LED_with_PWM_2
```

Check the PWM tutorial for more information about how to set up the experiment etc.

If you want to use **roslaunch** instead of **rosrun**, create a launch file.

```
mkdir ~/my_first_ws/src/my_first_package/launch/
```

Create launch file **sr_ronex_flash_LED_with_PWM_2.launch**, and place it inside the **launch** folder.

```
<launch>
  <node name="sr_ronex_flash_LED_with_PWM_2" pkg="my_first_package" type="sr_ronex_flash_LED_with_PWM
</launch>
```

Now you can use **roslaunch** to run the code.

```
roslaunch my_first_package sr_ronex_flash_LED_with_PWM_2.launch
```

1.5.5 Running RoNeX without superuser

To be able to run the RoNeX driver as a normal Linux user (no superuser privileges), we provide you with a handy script.

```
rosrun ros_ethercat_loop ethercat_grant
```

Note: Executing the above commands in a terminal window once will set file capabilities until you next update the ros_ethercat package, after which you will need to set them again.

1.5.6 Setting parameters from a launch file

Configuration parameters of a ronex module can also be set from a user-defined launchfile, by using the dynparam set option.

This is probably the most common way to set up the right parameters for a user environment.

The following example shows how to set the mode to "output" for the I/O pin number 2.

```
<launch>

...

<node pkg="dynamic_reconfigure" type="dynparam" name="dynparam_i2"
       args="set /ronex/general_io/0 input_mode_2 false" />

...

</launch>
```

This page contains examples of how to access and adjust various RoNeX settings, there are multiple links for tasks that can be completed in various ways.

1. **Access module Parameters** [Command Line | Python | C++] : These examples show how to access data relating to your RoNeX modules on the ROS parameter server, from within your program.
2. Using aliases with your RoNeX
3. Creating a package to interact with RoNeX
4. Running the RoNeX driver without being superuser
5. Changing RoNeX Configuration Parameters from a Launch file

This section explains how to get started with a RoNeX system, including setting up your computer, launching the drivers and a run through overall layout of the system. This information is universal, regardless of which functional modules you are using.

1.6 *Setting up your computer*

How to install Ubuntu, ROS and the RoNeX packages on your machine and configure them correctly.

1.7 *Launching the RoNeX driver*

How to launch the RoNeX driver (this process is universal, it will automatically detect the modules present on the bus and activate the appropriate interfaces).

1.8 *Overview of the system*

An overview of the interfaces the RoNeX system provides, regardless of which functional modules you have connected.

1.9 *Interacting With RoNeX*

There are four main methods to interact with RoNeX, this section will give a brief introduction to these methods.

1.10 *General Examples and Tutorials*

This section provides non module specific examples of interacting with your RoNeX system.

CHAPTER 2

GIO Module

2.1 GIO Module - System Overview

This page describes some of the key parameters, services and topics you'll need to interact with when using the RoNeX GIO module. The information here builds on that related to the general RoNeX system which can be found here.

The serial number of the sample RoNeX GIO module used here was 12. On other systems the number 12 will be replaced with the serial number corresponding to the connected GIO module.

2.1.1 Topics

ROS Topics provide a fast, flexible way to transfer data between nodes.

Topics can be published or subscribed to from the command line or within a program. For more information see the tutorials.

- `/ronex/general_io/12/command/digital/0` *(0-11) [std_msgs/Bool]*

By publishing a Boolean value to one of these topics, you can turn the corresponding digital output on or off (if it has been configured as an ouput using the input_mode parameter).

- `/ronex/general_io/12/command/pwm/0` *(0-5) [sr_ronex_msgs/PWM]*

Publishing to one of these topics configures the corresponding PWM module. There are 6 modules in total, each controlling 2 channels. You can configure the period for each module, and the on time for each channel individually.

- `/ronex/general_io/12/parameter_updates`

RoNeX publishes information on the current configuration of the module i/o channels to this topic, including input_mode, clock divider and PWM period. As the name suggests, parameter updates will be reflected in the data published on this topic.

- `/ronex/general_io/12/parameter_descriptions`

RoNeX publishes a short description of the function of each of the module parameters to this topic.

- `/ronex/general_io/12/state`

This topic contains the current state of all of the channels on the RoNeX module, including digital channel values and input_mode configuration, PWM clock divider and analogue input values. If you're looking to monitor the activities of your RoNeX module, this is the place to get the information.

2.1.2 Parameters

The parameter server acts as a database of information to be shared between various nodes running within your ROS system.

Any of these parameters can be read using the rosparam get command. Module parameters such as input modes or click dividers are dynamic parameters, and as such you need to **use the dynamic reconfigure** interfaces listed in the changing the configuration section of the tutorials in order for them to update correctly.

- `/ronex/general_io/12/input_mode_0` *(0-11)*

These parameters (one for each digital io channel) contain the current mode of each digital channel, True for input mode, and False for output mode.

- `/ronex/general_io/12/pwm_clock_divider`

The master clock divider, used in configuring PWM output parameters. For more information on using PWM functionality see the tutorials.

- `/ronex/general_io/12/pwm_period_0` *(0-5)*

The cycle period for each PWM unit. For more information on using PWM functionality see the tutorials.

2.1.3 Services

ROS Services provide a reliable communication method for operations that require a response or confirmation, such as the updating of system configuration.

Any of these services can be called directly from the command line using rosservice call, or from a program as described here.

- `/ronex/general_io/12/set_parameters`

This service allows for dynamic reconfiguration of the module parameters listed above. Usage can be found in the tutorials.

2.2 GIO Tutorials

2.2.1 GIO - Read analogue inputs (command line)

Viewing GIO analogue input data on the command line is incredibly simple. The rostopic echo command subscribes to the topic in question then displays an data received in the terminal window:

```
rostopic echo /ronex/general_io/12/state
```

This will display the status of the whole GIO module, if you'd prefer to display a single analogue channel's data instead (channel 0 in the example below) you can use the following command:

```
rostopic echo /ronex/general_io/12/state/analogue[0]
```

2.2.2 GIO - Read analogue inputs (Gui)

The best way to analyse data from an analogue input channel in the GUI is to use the Plot plugin, which will allow you to display the data in a graphical manner.

Once you've opened the Plot plugin you can then enter the name of the topic you want to display, the topic name will auto complete, but you'll have to add the name of the data field on the end manually (e.g. analogue[0]).

2.2.3 GIO - Read analogue inputs (python)

A General I/O Module consists of one GIO Node board and one GIO Peripheral board. Each module has 12 Analogue sampling channels (12bits, 1kHz) (a GIO Node board has 6 analogue input channels, a GIO Peripheral board has also 6).

This example demonstrates how to read analogue data with RoNeX. As input, you can use for example a potentiometer (a three-terminal resistor with a sliding contact that forms an adjustable voltage divider).

2.2.4 The code

First change directories to your **sr_ronex_examples** package.

```
$ roscd sr_ronex_examples/
```

Python file **sr_ronex_read_analog_data.py** is located inside the **src** directory.

```
#!/usr/bin/env python

import rospy
from sr_ronex_msgs.msg import GeneralIOState

def generalIOState_callback(data):
    analogue = data.analogue
    rospy.loginfo( "analogue = %s", analogue )

if __name__ == "__main__":
    rospy.init_node("sr_ronex_read_analog_data")
    topic = "/ronex/general_io/12/state"
    rospy.Subscriber( topic, GeneralIOState, generalIOState_callback )
    rospy.spin()
```

The Code Explained

```
import rospy
from sr_ronex_msgs.msg import GeneralIOState
```

First we load import appropriate modules and messages, including rospy to make us a ros node and get access to the topics, and the GeneralIOState message format we're going to receive from the state topic.

```
if __name__ == "__main__":
    rospy.init_node("sr_ronex_read_analog_data")
    topic = "/ronex/general_io/12/state"
    rospy.Subscriber( topic, GeneralIOState, generalIOState_callback )
    rospy.spin()
```

In the main function (at the bottom of the file) we initialise the ROS node that will publish the messages, and set up the topic to be subscribed to. We then create the subscriber specifying topic, message format and callback function, then finally the spin() function is called to start listening for messages.

For simplicity, in this example we hard code the ronex id (12), and you will need to change it to correspond to that of your RoNeX GIO module. To make this example more robust, the path to the first connected device could be retrieved from the parameter server. To learn how to do this you can follow the [[Parse Parameter Server (Python) Tutorial|Parse Parameter Server (Python)]].

```
def generalIOState_callback(data):
    analogue = data.analogue
    rospy.loginfo( "analogue = %s", analogue )
```

This callback function is called every time a message is received on the state topic. We extract the analogue data from the state message and then output it to screen. The analogue data is in the format tuple.

2.2.5 Running the code

First make sure that the RoNeX driver is running (see Launch driver).

Once this is done we can run our Python script:

```
rosrun sr_ronex_examples sr_ronex_read_analog_data.py
```

Now you should to able to see the analogue data on the console.

2.2.6 GIO - Read analogue inputs (C++)

A General I/O Module consists of one GIO Node board and one GIO Peripheral board. Each module has 12 Analogue sampling channels (12bits, 1kHz) (a GIO Node board has 6 analogue input channels, a GIO Peripheral board has also 6).

This example demonstrates how to read analogue data with RoNeX. As input, you can use for example a potentiometer (a three-terminal resistor with a sliding contact that forms an adjustable voltage divider).

The code

Change directories to your sr_ronex_examples package.

```
roscd sr_ronex_examples/
```

C++ file **sr_ronex_read_analog_data.cpp** is located inside the **src** directory.

```
#include <string>
#include <ros/ros.h>
#include "sr_ronex_msgs/GeneralIOState.h"

void generalIOState_callback(const sr_ronex_msgs::GeneralIOState::ConstPtr& msg)
{
  const std::vector<short unsigned int> &analogue = msg->analogue;
  const size_t len = analogue.size();
  for (size_t k = 0; k < len; k++)
    ROS_INFO_STREAM( "analogue[" << k << "] = " << analogue[k] << "\n" );
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "sr_ronex_read_analog_data");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe( "/ronex/general_io/12/state",
                                     1000,
                                     generalIOState_callback);
  ros::spin();
}
```

The Code Explained

```
#include <string>
#include <ros/ros.h>
#include "sr_ronex_msgs/GeneralIOState.h"
```

First we load import appropriate modules and messages, including the standard ros.h header to create a ros node and get access to the topics, and the GeneralIOState message format we're going to receive from the state topic.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "sr_ronex_read_analog_data");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe( "/ronex/general_io/12/state", 1000, generalIOState_callback);
  ros::spin();
}
```

In the main function (at the bottom of the file) we initialise the ROS node that will publish the messages, and add a handle to this node. We then create the subscriber specifying topic, message queue size and callback function, then finally the spin() function is called to start listening for messages.

For simplicity, in this example we hard code the ronex id (12), and you will need to change it to correspond to that of your RoNeX GIO module. To make this example more robust, the path to the first connected device could be retrieved from the parameter server. To learn how to do this you can follow the Parse Parameter Server (CPP) Tutorial.

```
void generalIOState_callback(const sr_ronex_msgs::GeneralIOState::ConstPtr& msg)
{
  const std::vector<short unsigned int> &analogue = msg->analogue;
  const size_t len = analogue.size();
  for (size_t k = 0; k < len; k++)
    ROS_INFO_STREAM( "analogue[" << k << "] = " << analogue[k] << "\n" );
}
```

This callback function is called every time a message is received on the state topic. We first create a vector of type short unsigned int then populate it with the analogue values from the current message, and assign the length of this vector to another variable. We then cycle through the vector outputting the values to the screen and ROS log files.

Running the code

If you're running this code from your own workspace, you'll first need to build it using Catkin, if you're not sure how to do this you can follow the instructions here.

Next, make sure that a roscore and the RoNeX driver are running (see Launch driver).

Digital i/o channel 0 needs to be configured as an output in order to flash the LED (all digital channels are set to input by default). The easiest way to do this is to use the dynamic reconfigure GUI. and set `input_mode_0` to `false`.

Once this is done we can run our C++ program:

```
rosrun sr_ronex_examples sr_ronex_read_analog_data
```

Now you should to able to see the analogue data on the console.

2.2.7 Flashing a LED (command line)

ROS Terminal commands are a quick and easy way to see what's going on with your RoNeX module. If you need to run something over and over again you'll want to write a script or use the GUI, but for one off tests, or when you only have command line access, these commands can be very handy.

For the following commands you can use the tab key to auto complete the name of the topic/service and message type, and once you've entered the message type you can actually double tap tab again to produce an empty message in the correct format, handy for when you're not familiar with the message types.

So if we're interfacing with an LED we need to set the corresponding digital channel as an output, this can be done from the command line by calling the set_parameters service. In the example below we have also set the pwm_clock_divider parameter in the same message for convenience.

```
rosservice call /ronex/general_io/12/set_parameters "config:
  bools:
  - {name: 'input_mode_0', value: false}
  ints:
  - {name: 'pwm_clock_divider', value: 20}"
```

Once the channel is configured correctly we can publish a message to the command topic for the corresponding PWM module as shown below. Changing the pwm_on_time_0 value will adjust the brightness of the LED.

```
rostopic pub /ronex/general_io/12/command/pwm/0 sr_ronex_msgs/PWM "pwm_period: 64000
    pwm_on_time_0: 1000
    pwm_on_time_1: 0"
```

2.2.8 Flashing a LED (GUI)

The ROS GUI contains a number of useful graphical tools which can be used to interact with your GIO module. In this example we will look at how to configure a PWM module to control an LED on digital channel 0.

The first thing we need to do is set channel 0 as an output, which can be done by unchecking the appropriate box in the Dynamic Reconfigure GUI.

We can then open the publisher plugin, select the topic and message type, then configure the message. Not that the box on the left hand side must be ticked in order to publish the message.

2.2.9 Flashing a LED (python)

This example demonstrates how to flash an LED using PWM (Pulse Width Modulation) by varying duty cycle (i.e., the amount of time in the period that the pulse is active or high).

Note that General I/O (GIO) module consists of one GIO Node board and one GIO Peripheral board, so there are 12 digital I/O channels (a GIO Node board has 6 digital I/O channels, a GIO Peripheral board has also 6 digital I/O channels). PWM is available on all 12 digital channels.

For this example, we assume that the LED is connected to digital channel 0.

The code

First change directories to your **sr_ronex_examples** package.

```
roscd sr_ronex_examples/
```

Python file **sr_ronex_flash_LED_with_PWM.py** is located inside the **src** directory.

```
#!/usr/bin/env python

import rospy
from sr_ronex_msgs.msg import PWM
```

```
#-------------------------------------------------------------------------------

def flashLED(topic):
    pwm_period = 320
    pwm_on_time_0 = pwm_period
    pwm_on_time_1 = 0

    pub = rospy.Publisher( topic, PWM )
    while not rospy.is_shutdown():
        pwm_on_time_0 -= 10
        if pwm_on_time_0 < 0:
            pwm_on_time_0 = pwm_period
        pwm = PWM()
        pwm.pwm_period    = pwm_period
        pwm.pwm_on_time_0 = pwm_on_time_0
        pwm.pwm_on_time_1 = pwm_on_time_1

        pub.publish( pwm )
        rospy.sleep( 0.01 )

#-------------------------------------------------------------------------------

if __name__ == "__main__":
    rospy.init_node('sr_ronex_flash_LED_with_PWM')

    topic = "/ronex/general_io/12/command/pwm/0"
    try:
        flashLED(topic)
    except rospy.ROSInterruptException:
        pass
```

The Code Explained

Now lets have a look at the code:

```
import rospy
from sr_ronex_msgs.msg import PWM
```

First we load import appropriate modules and messages, including rospy to make us a ros node and get access to the topics, sleep function etc and the PWM message format we're going to publish.

```
if __name__ == "__main__":
    rospy.init_node('sr_ronex_flash_LED_with_PWM')

    topic = "/ronex/general_io/12/command/pwm/0"
    try:
        flashLED(topic)
    except rospy.ROSInterruptException:
        pass
```

In the main function (at the bottom of the file) we initialise the ROS node that will publish the messages, set up the topic to be published to (we assume the LED is connected to digital channel 0), then call the function to flash the LED.

For simplicity, in this example we hard code the ronex id (12), and you will need to change it to correspond to that of your RoNeX GIO module. To make this example more robust, the path to the first connected device could be retrieved from the parameter server. To learn how to do this you can follow the Parse Parameter Server (Python) Tutorial.

```
def flashLED(topic):
    pwm_period = 320
    pwm_on_time_0 = pwm_period
    pwm_on_time_1 = 0
    pub = rospy.Publisher( topic, PWM )
```

The name of the topic to be published to is passed to the flashLED function from the main function. We set the PWM period to 320, and then the on time for channel 0 to the same value. This means corresponds to a 100% duty cycle, meaning the LED will receive full power. Channel 1 is not used, so we set the on time to 0. The publisher is then initialised, with the topic name passed from main, and message format PWM.

```
while not rospy.is_shutdown():
    pwm_on_time_0 -= 10
    if pwm_on_time_0 < 0:
        pwm_on_time_0 = pwm_period
```

Next we have a while loop that runs continuously until ROS is shutdown (or the program is interrupted). For every increment of the loop we subtract 10 from the channel 0 on time, making the LED gradually dimmer. If the on time has reached 0 (i.e. the LED is completely off), we set it equal to PWM period again (full power).

```
pwm = PWM()
pwm.pwm_period    = pwm_period
pwm.pwm_on_time_0 = pwm_on_time_0
pwm.pwm_on_time_1 = pwm_on_time_1

pub.publish( pwm )
rospy.sleep( 0.01 )
```

We then create a PWM message and populate it with the values we have just assigned, publish it (send the command) and sleep for 10ms before returning to the start of the while loop.

Running the code

First make sure that the RoNeX driver is running (see Launch driver).

Digital i/o channel 0 needs to be configured as an output in order to flash the LED (all digital channels are set to input by default). The easiest way to do this is to use the GUI and set `input_mode_0` to `false`.

Once this is done we can run our Python script:

```
rosrun sr_ronex_examples sr_ronex_flash_LED_with_PWM.py
```

You should now see your LED flashing. You can try adjusting the pwm_on_time_0 increments and sleep time to achieve different light patterns.

2.2.10 Flashing a LED (C++)

This example demonstrates how to flash an LED using PWM (Pulse Width Modulation) by varying duty cycle (i.e., the amount of time in the period that the pulse is active or high).

Note that General I/O (GIO) module consists of one GIO Node board and one GIO Peripheral board, so there are 12 digital I/O channels (a GIO Node board has 6 digital I/O channels, a GIO Peripheral board has also 6 digital I/O channels). PWM is available on all 12 digital channels.

For this example, we assume that the LED is connected to digital channel 0.

The code

First change directories to your **sr_ronex_examples** package.

```
roscd sr_ronex_examples/
```

C++ file **sr_ronex_flash_LED_with_PWM.cpp** is located inside the **src** directory.

```
#include <ros/ros.h>
#include "sr_ronex_msgs/PWM.h"

void flash_LED( ros::NodeHandle& n, const std::string& topic )
{
  ros::Publisher pub = n.advertise<sr_ronex_msgs::PWM>( topic, 1000 );
  short unsigned int pwm_period = 320;
  short unsigned int pwm_on_time_0 = pwm_period;
  short unsigned int pwm_on_time_1 = 0;

  ros::Rate loop_rate(100);
  while ( ros::ok() )
  {
    pwm_on_time_0 -= 10;
    if (pwm_on_time_0 < 0)
      pwm_on_time_0 = pwm_period;
    sr_ronex_msgs::PWM msg;
    msg.pwm_period    = pwm_period;
    msg.pwm_on_time_0 = pwm_on_time_0;
    msg.pwm_on_time_1 = pwm_on_time_1;
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "sr_ronex_flash_LED_with_PWM");
  ros::NodeHandle n;
  std::string topic = "/ronex/general_io/12/command/pwm/0";
  flash_LED( n, topic );
  return 0;
}
```

The Code Explained

Now lets have a look at the code:

```
#include <ros/ros.h>
#include "sr_ronex_msgs/PWM.h"
```

First we load import appropriate modules and messages, including the standard ros.h header to create a ros node and get access to the topics, sleep function etc and the PWM message format we're going to publish.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "sr_ronex_flash_LED_with_PWM");
  ros::NodeHandle n;
  std::string topic = "/ronex/general_io/12/command/pwm/0";
```

```
  flash_LED( n, topic );
  return 0;
}
```

In the main function (at the bottom of the file) we initialise the ROS node that will publish the messages, and add a handle to this node. We then set up the topic to be published to (we assume the LED is connected to digital channel 0), then call the function to flash the LED.

For simplicity, in this example we hard code the ronex id (12), and you will need to change it to correspond to that of your RoNeX GIO module. To make this example more robust, the path to the first connected device could be retrieved from the parameter server. To learn how to do this you can follow the Parse Parameter Server (C++) Tutorial.

```
void flash_LED( ros::NodeHandle& n, const std::string& topic )
{
  ros::Publisher pub = n.advertise<sr_ronex_msgs::PWM>( topic, 1000 );
  short unsigned int pwm_period = 320;
  short unsigned int pwm_on_time_0 = pwm_period;
  short unsigned int pwm_on_time_1 = 0;
```

The handle to the node we created in the main function and the name of the topic to be published to is passed to the flash_LED function from the main function. The publisher is then initialised, with message format PWM, the topic name passed from main, and a queue size of 1000. We set the PWM period to 320, and then the on time for channel 0 to the same value. This means corresponds to a 100% duty cycle, meaning the LED will receive full power. Channel 1 is not used, so we set the on time to 0.

```
ros::Rate loop_rate(100);
while ( ros::ok() )
{
  pwm_on_time_0 -= 10;
  if (pwm_on_time_0 < 0)
    pwm_on_time_0 = pwm_period;
```

Next we create a rate variable which will be used in combination with a sleep to maintain a 100Hz rate for the following loop. We then have a while loop that runs continuously until ROS is shutdown (or the program is interrupted). For every increment of the loop we subtract 10 from the channel 0 on time, making the LED gradually dimmer. If the on time has reached 0 (i.e. the LED is completely off), we set it equal to PWM period again (full power).

```
    sr_ronex_msgs::PWM msg;
    msg.pwm_period    = pwm_period;
    msg.pwm_on_time_0 = pwm_on_time_0;
    msg.pwm_on_time_1 = pwm_on_time_1;
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

We then create a PWM message and populate it with the values we have just assigned, publish it, call spinOnce() to send out the command, and sleep for the required time to maintain a 100Hz rate before returning to the start of the while loop.

Running the code

If you're running this code from your own workspace, you'll first need to build it using Catkin, if you're not sure how to do this you can follow the instructions *here </General/Create-a-package-to-interact-with-RoNeX#running-the-code>*.

Next sure that a roscore and the RoNeX driver are running (see Launch driver).

Digital i/o channel 0 needs to be configured as an output in order to flash the LED (all digital channels are set to input by default). The easiest way to do this is to use the GUI and set `input_mode_0` to `false`.

Once this is done we can run our C++ program:

```
rosrun sr_ronex_examples sr_ronex_flash_LED_with_PWM
```

You should now see your LED flashing. You can try adjusting the pwm_on_time_0 increments and sleep time to achieve different light patterns.

2.2.11 Configure GIO Module (command line)

To update the configuration of your GIO module, for example setting each channel as an input or output mode, you'll need to use the Dynamic Reconfigure interfaces. To do this from the command line we can call a service and tell the RoNeX driver which configurations we want to change.

For the following command you can use the tab key to auto complete the name of the service and message type, and once you've entered the message type you can actually double tap tab again to produce an empty message in the correct format, handy for when you're not familiar with the message types.

In the example below we will set 4 of the digital channels as an outputs:

```
rosservice call /ronex/general_io/12/set_parameters "config:
     bools:
     - {name: '', value: false}
     ints:
     - {name: '', value: 0}
     strs:
     - {name: '', value: ''}
     doubles:
     - {name: '', value: 0.0}
     groups:
     - {name: '', state: false, id: 0, parent: 0}"
```

The service call will return the current state of the GIO parameters, alternatively you can view this information at any time by echoing the relevant topic:

```
rostopic list /ronex/general_io/12/parameter_updates
```

2.2.12 Configure a ronex module using the GUI

- Run the dynamic reconfigure GUI directly:

```
rosrun rqt_reconfigure rqt_reconfigure
```

- Or load it through the main GUI:

```
rosrun rqt_gui rqt_gui
```

ant then open the Dynamic Reconfigure plugin: Plugins->Dynamic Reconfigure

- Select the ronex device you want to configure from the list on the left.
- Tweak the parameters:
- Via the sliders
- By editing them (in that case you need to click outside the text box or press enter for the new value to take effect)

2.2.13 Configure GIO Module (python)

This tutorial will walk you through the basics of using the dynamic reconfigure client in a python script to adjust your RoNeX configuration. If you are often executing different RoNeX scripts that require different module configuration, it may be beneficial adjust the configuration at the start of each script.

This example can be found under sr_ronex_examples/src/change_ronex_configuration.py # The Code

```
#!/usr/bin/env python

import rospy
import dynamic_reconfigure.client

class ChangeRonexConfigurationExample(object):

    def __init__(self):
        ronex_id = "test_ronex"
        ronex_path = "/ronex/general_io/" + ronex_id + "/"
        self.configure_ronex(ronex_path)

    def configure_ronex(self, path):
        client = dynamic_reconfigure.client.Client(path)
        params = { 'input_mode_0' : False, 'input_mode_1' : False, 'pwm_period_0' : 200 , 'pwm_clock_
        config = client.update_configuration(params)

if __name__ == "__main__":
    rospy.init_node("change_ronex_configuration_py")
    ChangeRonexConfigurationExample()
```

The Code Explained

Now lets break down the code to make it easier to reuse in other programs:

```
#!/usr/bin/env python

import rospy
import dynamic_reconfigure.client
```

We need to import the dynamic reconfigure client module at the start of the script.

```
if __name__ == "__main__":
    rospy.init_node("change_ronex_configuration_py")
    ChangeRonexConfigurationExample()
```

At the bottom of the script we have the main function where we initialise a node to carry out the config changes. If you've already created a node in your script for other purposes, you can just use that one. We then call the ChangeRonexConfigurationExample class which will execute the config changes.

```
def __init__(self):
    ronex_id = "test_ronex"
    ronex_path = "/ronex/general_io/" + ronex_id + "/"
    self.configure_ronex(ronex_path)
```

The class' init function gets called automatically. Here we set the path to the RoNeX module for which we want to change the configuration. This example assumes the alias "test_ronex" has already been set for your module, as described in this tutorial. Alternatively you can just use the serial number of your RoNeX module here instead. This path is then passed to the configure_ronex function.

```
def configure_ronex(self, path):
    client = dynamic_reconfigure.client.Client(path)
    params = { 'input_mode_0' : False, 'input_mode_1' : False, 'pwm_period_0' : 200 , 'pwm_clock_divi
    config = client.update_configuration(params)
```

Here we create a client instance and tell it the path to the RoNeX module to configure. Next we list the parameters than we want to change, here we are setting the first digital I/O channel to output mode,the second channel to input mode, adjusting pwm_period_0 and the PWM clock divider. We then call update_configuration to pass these changes to the dynamic reconfigure server.

The contents of this configure_ronex function (plus the dynamic reconfigure client module import from the top of the script) are all you really need if you want change your configuration from an existing script.

Running the code

Make sure that a roscore is up and running:

```
roscore
```

Then run the driver (see Launch driver).

Now we can execute the example script:

```
rosrun sr_ronex_examples change_ronex_configuration.py
```

Now if you echo the contents of the parameter_descriptions topic for this module, you should see that the configuration has been updated accordingly.

2.2.14 Configure GIO Module (c++)

This tutorial will walk you through the basics of calling the RoNeX /set_parameters service in a C++ script to adjust your RoNeX configuration. If you are often executing different RoNeX scripts that require different module configuration, it may be beneficial adjust the configuration at the start of each script.

This example can be found under `sr_ronex_examples/src/change_ronex_configuration.cpp`

The Code

```
#include <ros/ros.h>
#include <ros/console.h>
#include <string>

#include <dynamic_reconfigure/BoolParameter.h>
#include <dynamic_reconfigure/IntParameter.h>
#include <dynamic_reconfigure/Reconfigure.h>
#include <dynamic_reconfigure/Config.h>

class ChangeRonexConfigurationExample
{
public:

  ChangeRonexConfigurationExample()
  {
  }

  ~ChangeRonexConfigurationExample()
```

```
  {
  }

  void configureRonex(std::string path)
  {
    dynamic_reconfigure::ReconfigureRequest srv_req;
    dynamic_reconfigure::ReconfigureResponse srv_resp;
    dynamic_reconfigure::BoolParameter bool_param;
    dynamic_reconfigure::IntParameter int_param;
    dynamic_reconfigure::Config conf;

    bool_param.name = "input_mode_0";
    bool_param.value = false;
    conf.bools.push_back(bool_param);

    bool_param.name = "input_mode_1";
    bool_param.value = false;
    conf.bools.push_back(bool_param);

    int_param.name = "pwm_period_0";
    int_param.value = 200;
    conf.ints.push_back(int_param);

    int_param.name = "pwm_clock_divider";
    int_param.value = 3000;
    conf.ints.push_back(int_param);

    srv_req.config = conf;

    ros::service::call(path + "/set_parameters", srv_req, srv_resp);
  }

};

int main(int argc, char **argv)
{
  ros::init(argc, argv, "change_ronex_configuration_cpp");
  ChangeRonexConfigurationExample example;
  std::string ronex_id = "test_ronex";
  std::string ronex_path = "/ronex/general_io/" + ronex_id + "/";
  example.configureRonex(ronex_path);
  return 0;
}
```

The Code Explained

Now lets break down the code into manageable bite sized portions and explain each piece:

```
#include <ros/ros.h>
#include <ros/console.h>
#include <string>

#include <dynamic_reconfigure/BoolParameter.h>
#include <dynamic_reconfigure/IntParameter.h>
#include <dynamic_reconfigure/Reconfigure.h>
#include <dynamic_reconfigure/Config.h>
```

First we import the standard ROS headers, plus those for the dynamic reconfigure formats we will be using.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "change_ronex_configuration_cpp");
  ChangeRonexConfigurationExample example;
  std::string ronex_id = "test_ronex";
  std::string ronex_path = "/ronex/general_io/" + ronex_id + "/";
  example.configureRonex(ronex_path);
  return 0;
}
```

At the bottom of the script we have the main function where we first initialise a node to carry out the config changes, then create an instance of the ChangeRonexConfigurationExample class. The path to the RoNeX module of which we want to configure is then defined, using the "test_ronex" alias, which has been defined using the method described in this tutorial. Alternatively you can just use the serial number of your RoNeX module here instead.

We then call the `configureRonex` function from the instance of the `ChangeRonexConfigurationExample` class.

```
void configureRonex(std::string path)
{
  dynamic_reconfigure::ReconfigureRequest srv_req;
  dynamic_reconfigure::ReconfigureResponse srv_resp;
  dynamic_reconfigure::BoolParameter bool_param;
  dynamic_reconfigure::IntParameter int_param;
  dynamic_reconfigure::Config conf;
```

At the start of the configureRonex function we define the variables that we will be using to construct the service call message. The request will contain the config which in turn will contain the bool and int parameters. The response will contain the data returned by the service.

```
bool_param.name = "input_mode_1";
bool_param.value = false;
conf.bools.push_back(bool_param);

int_param.name = "pwm_period_0";
int_param.value = 200;
conf.ints.push_back(int_param);
```

Next we populate the config message with the various parameters. The section of code above shows how we set the name and value for bool and int parameters and push them into the config variable. Here we set configure digital I/O channel 1 as an output, and the period for PWM module 0 to 200.

```
srv_req.config = conf;
ros::service::call(path + "/set_parameters", srv_req, srv_resp);
```

Finally we populate the request variable with the config, and call the set_parameters service with the request and response variables as arguments.

2.2.15 Running the code

Make sure that a roscore is up and running:

```
roscore
```

Then run the driver (see Launch driver).

Now we can execute the example script:

```
rosrun sr_ronex_examples change_ronex_configuration
```

Now if you echo the contents of the parameter_descriptions topic for this module, you should see that the configuration has been updated accordingly.

2.2.16 Control a Joint Based Robot

If you're working with a joint based robot, thanks to RoNeX, you can control your whole robot with just a few lines of xml (simply adding custom transmissions to the URDF robot model).

You can find this example in the **sr_ronex_examples** package. The files used are:

- *model/ronex.urdf*: a description of the robot
- *launch/sr_ronex_urdf.launch*: a launch file to start the drivers, the controllers, and a few utils.
- *config/joint_position_controller.yaml*: the parameter settings.

Set an alias for your RoNeX module

To be able to run this example, you'll need to setup an alias for your RoNeX module. To do this, simply follow [[the instruction|Using-aliases-with-your-RoNeX]].

Creating your URDF

There are extensive Tutorials on the ROS wiki, covering how to write a URDF. We'll assume you're familiar with URDF for this tutorial.

Let's use this very simple robot model as a base, with one continuous joint.

```
<?xml version="1.0"?>
<robot name="flexible">
  <!-- standard urdf -->
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
      <material name="blue">
        <color rgba="0 0 .8 1"/>
      </material>
    </visual>
  </link>

  <link name="head">
    <visual>
      <geometry>
        <sphere radius="0.2"/>
      </geometry>
      <material name="white"/>
    </visual>
  </link>

  <joint name="head_swivel" type="continuous">
    <parent link="base_link"/>
    <child link="head"/>
```

```
    <axis xyz="0 0 1"/>
    <origin xyz="0 0 0.3"/>
  </joint>
</robot>
```

Let's say that a **position sensor** is plugged into the **analogue pin 6** of your General I/O board. We want to map this position sensor to our robot's joint position. To do this, we use a **RonexTransmission**. Add the following after the `</joint>` in the previous URDF:

```
<transmission type="sr_ronex_transmissions/RonexTransmission" name="my_ronex_transmission">
  <joint name="head_swivel"/>
  <mapping ronex="/ronex/general_io/test_ronex" property="position" analogue_pin="6" scale="0.001" of
</transmission>
```

The first line loads the transmission and gives it the name *my_ronex_transmission*. As you can see, this transmission is acting on the joint *head_swivel* (which has been defined previously in the URDF). The **mapping** line is then mapping one given RoNeX module (using it's RoNeX id), to the *position* property of our joint. We're specifying which analogue pin to use (6 in this example), and also scaling the input to a better range by using a scale and offset.

Let's now assume that you've also got a **servo**. It is plugged into the **PWM module 1 - pin 0** using **digital pin 0** as the direction pin. We now want to control the joint in position. We can refine the previous transmission to add this servo.

```
<transmission type="sr_ronex_transmissions/RonexTransmission" name="my_ronex_transmission">
  <joint name="head_swivel"/>
  <mapping ronex="/ronex/general_io/test_ronex" property="position" analogue_pin="6" scale="0.001" of
  <mapping ronex="/ronex/general_io/test_ronex" property="command" pwm_module="1" pwm_pin="0" directi
</transmission>
```

As you can see we're adding a *command* mapping line to map the *command_* of the joint to the servo plugged into our PWM module.

If you have an effort sensor (on analogue pin 7 for example), you could also map it in the transmission:

```
<transmission type="sr_ronex_transmissions/RonexTransmission" name="my_ronex_transmission">
  <joint name="head_swivel"/>
  <mapping ronex="/ronex/general_io/test_ronex" property="position" analogue_pin="6" scale="0.001" of
  <mapping ronex="/ronex/general_io/test_ronex" property="effort"   analogue_pin="7" scale="1.0" offs
  <mapping ronex="/ronex/general_io/test_ronex" property="command" pwm_module="1" pwm_pin="0" directi
</transmission>
```

As you can see each mapping line is optional so you can adjust the transmission to your available hardware.

Writing the launch file

Now that you have a model of your robot which maps the different sensors and motors you have to your RoNeX's inputs, we want to load the robot model, start the driver, and publish the current joint states (position, effort, velocity) of our bot. To do this you can use this simple launch file.

```
<launch>
  <!-- Load the robot description -->
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find sr_ronex_examples)/model/ro

  <!-- Allows to specify the ethernet interface to be used. It defaults to the value of the env var
  <arg name="ethercat_port" default="$(optenv ETHERCAT_PORT eth0)" />

  <!-- Start the ronex driver -->
  <node name="ronex" pkg="pr2_ethercat" type="pr2_ethercat" args="-i $(arg ethercat_port) -r /robot_d
```

```
  <!-- publishes the joint states -->
  <include file="$(find ros_ethercat_model)/launch/joint_state_publisher.launch"/>
</launch>
```

You can view your robot using rviz. If you add a *Robot Model* plugin and use */base_link* as the fixed frame you should see a D2R2 like robot. You can swivel the head around using your analogue "position" sensor.

Adding Controllers

Now that we can both read the position and control a servo, let's start some joint position controllers. To do this, we first need to make the joint *head_swivel* controllable. We'll use the RoNeX fake calibration controllers for that.

Setting up the different controller settings

If you create a *joint_position_controller.yaml* file you can define these simple parameters.

```
head_swivel_fake_calib:
  type: sr_ronex_controllers/FakeCalibrationController
  joint: head_swivel
```

Now that it is possible to use our joints in the standard ROS controller, we can setup a PID joint position controller on our head swivel topic. Let's add the parameters to our yaml controller parameter file. We can use the ROS standard robot_mechanism_controllers/JointPositionController.

```
head_swivel_controller:
  type: robot_mechanism_controllers/JointPositionController
  joint: head_swivel
  pid: &head_swivel_gains
    p: 1000.0
    d: 0.0
    i: 0.0
    i_clamp: 0.0
```

Loading the settings and spawning the controllers

The different settings for our controllers are ready to be loaded. We can go back to editing the launch file to load them and then we'll spawn the fake calibration controllers (making the joints controllable) and the position controller.

```
<!-- Loads the controller parameter -->
  <rosparam command="load" file="$(find sr_ronex_examples)/config/joint_position_controller.yaml" />

  <!-- spawn fake calibration controller: the pr2 controllers need the joints
       to be set to calibrated = true to work -->
  <node name="fake_calib_controllers_spawner"
        pkg="pr2_controller_manager" type="spawner" output="screen"
        args="head_swivel_fake_calib" />

  <!-- spawning traditional joint controllers -->
  <node name="joint_controllers_spawner"
        pkg="pr2_controller_manager" type="spawner" output="screen"
        args="--wait-for=calibrated head_swivel_controller" />

  <!-- publishes tf from joint states to be able to view in rviz -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>
```

Making sure our pins are set to output mode

The last thing we need to do is to change the mode of the pins we use for the servo to **output**. By default all the digital pins are set to input. To do this we access the dynamic reconfigure interface from our launch file.

```
<!-- setting the corresponding pins to output mode on the RoNeX -->
<node pkg="dynamic_reconfigure" type="dynparam" name="dynparam_i2"
  args="set /ronex/general_io/test_ronex input_mode_2 false" />
<node pkg="dynamic_reconfigure" type="dynparam" name="dynparam_i5"
  args="set /ronex/general_io/test_ronex input_mode_5 false" />
```

2.2.17 Accessing the RoNeX data directly from a controller

This tutorial shows how you can access RoNex data directly from a controller at 1kHz.

As input, we use here a potentiometer (a three-terminal resistor with a sliding contact that forms an adjustable voltage divider). And it is connected to the first analogue sampling channel (i.e., index 0) on the General I/O module with the alias "test_ronex".

Files belonging to this tutorial can be found in the **sr_ronex_examples** package.

- *model/simple_robot.urdf*: a description of the robot.
- *launch/sr_ronex_simple_controller.launch*: a launch file to start the drivers, the controllers, and a few utils.
- *config/data_access_controller.yaml*: the parameter settings.
- *controller_plugins.xml*: the plugin settings.

For a detailed desription of how these files are used, please read this tutorial: Working with a joint based robot

The code

Change directories to your **sr_ronex_examples** package.

```
roscd sr_ronex_examples/
cd src
pwd
```

C++ header file **sr_ronex_simple_controller.hpp** is located inside the **include/sr_ronex_examples** directory.

```
#pragma once

#include <ros/node_handle.h>

#include <controller_interface/controller.h>
#include <ros_ethercat_model/robot_state.hpp>
#include <sr_ronex_hardware_interface/mk2_gio_hardware_interface.hpp>
#include <realtime_tools/realtime_publisher.h>
#include <sr_ronex_utilities/sr_ronex_utilities.hpp>

#include <std_msgs/Bool.h>
#include <sr_ronex_msgs/PWM.h>

namespace ronex
{
  class SrRoNeXSimpleController
    : public controller_interface::Controller<ros_ethercat_model::RobotState>
```

```
  {
public:
    SrRoNeXSimpleController();
    virtual ~SrRoNeXSimpleController();

    virtual bool init(ros_ethercat_model::RobotState* robot, ros::NodeHandle &n);

    virtual void starting(const ros::Time&);

    virtual void update(const ros::Time&, const ros::Duration&);

    virtual void stopping(const ros::Time&);

  private:
    int loop_count_;

    ronex::GeneralIO* general_io_;
  };
}
```

C++ source file **sr_ronex_simple_controller.cpp** is located inside the **src** directory.

```
#include "sr_ronex_examples/sr_ronex_simple_controller.hpp"
#include "pluginlib/class_list_macros.h"

PLUGINLIB_EXPORT_CLASS( ronex::SrRoNeXSimpleController, controller_interface::ControllerBase)

namespace ronex
{
SrRoNeXSimpleController::SrRoNeXSimpleController()
  : loop_count_(0)
{
}

SrRoNeXSimpleController::~SrRoNeXSimpleController()
{
}

bool SrRoNeXSimpleController::init(ros_ethercat_model::RobotState* robot, ros::NodeHandle &n)
{
  assert (robot);

  std::string path("/ronex/general_io/test_ronex");
  general_io_ = static_cast<ronex::GeneralIO*>( robot->getCustomHW(path) );
  if( general_io_ == NULL)
  {
    ROS_ERROR_STREAM("Could not find RoNeX module (i.e., test_ronex). The controller is not loaded.")
    return false;
  }

  return true;
}

void SrRoNeXSimpleController::starting(const ros::Time&)
{
  // Do nothing.
}

void SrRoNeXSimpleController::update(const ros::Time&, const ros::Duration&)
```

```
{
  double position = general_io_->state_.analogue_[0];
  if (loop_count_++ % 100 == 0)
  {
    ROS_INFO_STREAM( "Position = " << position );
    loop_count_ = 0;
  }
}

void SrRoNeXSimpleController::stopping(const ros::Time&)
{
  // Do nothing.
}

}
```

The Code Explained

First, we look for General I/O module "test_ronex".

```
bool SrRoNeXSimpleController::init(ros_ethercat_model::RobotState* robot, ros::NodeHandle &n)
{
  assert (robot);

  std::string path("/ronex/general_io/test_ronex");
  general_io_ = static_cast<ronex::GeneralIO*>( robot->getCustomHW(path) );
  if( general_io_ == NULL)
  {
    ROS_ERROR_STREAM("Could not find RoNeX module (i.e., test_ronex). The controller is not loaded.");
    return false;
  }

  return true;
}
```

If the module is found, we repeatedly read the data from the first analogue sampling channel and output it to the console.

```
void SrRoNeXSimpleController::update(const ros::Time&, const ros::Duration&)
{
  double position = general_io_->state_.analogue_[0];
  if (loop_count_++ % 100 == 0)
  {
    ROS_INFO_STREAM( "Position = " << position );
    loop_count_ = 0;
  }
}
```

Note that class SrRoNeXSimpleController inherits from the template class controller_interface::Controller

```
namespace ronex
{
  class SrRoNeXSimpleController
    : public controller_interface::Controller<ros_ethercat_model::RobotState>
  {
  public:
    SrRoNeXSimpleController();
    virtual ~SrRoNeXSimpleController();
```

Running the code

First, if you are using **eth0** instead of **eth1**, please update the following line in the launch file.

```
<arg name="ethercat_port" default="$(optenv ETHERCAT_PORT eth1)" />
```

Now, run terminal in **sudo** mode.

```
sudo -s
```

Launch the controller (it was compiled to a library: **libsr_ronex_simple_controller.so**).

```
roslaunch sr_ronex_examples sr_ronex_simple_controller.launch
```

Now you should to able to see the analogue data on the console.

```
[ INFO] [1380207765.910351751]: Position = 4095
[ INFO] [1380207765.911347629]: Position = 4095
[ INFO] [1380207765.912353727]: Position = 4095
[ INFO] [1380207765.913331810]: Position = 4095
[ INFO] [1380207765.914350895]: Position = 4095
[ INFO] [1380207765.915329345]: Position = 4095
```

2.2.18 Interfacing Video Code

Analogue to PWM Configuration

This tutorial explains the calculations for converting an analogue value from a potentiometer into a PWM value corresponding to the same angle for a servo motor, as described in this tutorial and video: Youtube Interfacing Code

From the range of rotation of our potentiometer and resolution of the RoNeX ADC we know:

```
Potentiometer rotational range = 0 - 300°
12 bit Analogue input channel range = 0 - 4095
Analogue value at 180° = 4096 / 300 * 180 = 2457.6
```

From the Servo Datasheet we know:

```
Servo works on a 20ms control period.
On time of 0.5ms corresponds to an angle of 0°.
0.5ms on time in 20ms period corresponds to 2.5% duty cycle.
On time of 2.5ms corresponds to an of 180°
2.5ms on time in 20ms period corresponds to 12.5% duty cycle.
```

From the RoNeX Manual we know:

```
RoNeX base clock frequency  = 64MHz.
Master clock divider = 20.
Resultant master clock frequency = 64MHz/20 = 3.2 MHz.
PWM Period = 64000
Output frequency = 3.2MHz/64000 = 50 Hz
50Hz corresponds to period of 20ms.
```

From this we can deduce that:

```
In a 64000 period, 2.5% duty cycle = 64000 * 0.025 = 1600.
In a 64000 period, 12.5% duty cycle = 64000 * 0.125 = 8000.
```

For ease of calculation we offset the pwm_on_time by -1600 to give a range of 0-6400. This gives rise to the following conversion constant:

```
PWM range / Analogue range  = 6400 / 2457.6 = 2.6042
```

Which is then apparent in the Python code:

```
pwm.pwm_on_time_0 = int(2.6042*analogue[0] + 1600)
```

This Tutorial is to accompany the RoNeX Interfacing video on Youtube.

2.2.19 Startup

In the video we first start Roscore:

```
roscore
```

Then in a new terminal window elevate permissions to superuser (you can skip this step if you have followed the instructions in Ronex driver without sudo)

```
sudo -s
```

Then start the ROS driver:

```
roslaunch sr_ronex_launch sr_ronex.launch
```

Next in another new terminal we start the ROS GUI by running:

```
rosrun rqt_gui rqt_gui
```

Once in the GUI you can find the relevant plugins in the menu at the top. The topic we subscribe to in order to plot the analogue input data is:

```
/ronex/general_io/12/state/analogue[0]
```

The number 12 in the above command should be replaced by the serial number of your RoNeX GIO Module.

In the Dynamic Reconfigure plugin, untick the boxes relating to whichever input channels you plan to use as outputs (channel 0 in our case). For the PWM servo motor we used, the PWM clock divider was set to 20.

The topic we publish to in order to control the channel 0 PWM output is:

```
/ronex/general_io/12/command/pwm/0
```

Set pwm_period to 64000, then pwm_on_time_0 can be set from 1600 - 8000 to correspond to Servo angles of 0 - 180°.

2.2.20 The Code

The Python script to control the Servo with the Potentiometer shown in the video is as follows:

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('sr_ronex_examples')
import rospy
from sr_ronex_msgs.msg import PWM, GeneralIOState

def callback(data):
```

```
    analogue = data.analogue
    pwmTopic = '/ronex/general_io/12/command/pwm/0'
    pub = rospy.Publisher(pwmTopic, PWM)
    pwm = PWM()
    pwm.pwm_period = 64000
    pwm.pwm_on_time_0 = int(2.6042*analogue[0] + 1600)
    pwm.pwm_on_time_1 = 0
    pub.publish(pwm)

if __name__ == "__main__":

    rospy.init_node("ronex_pot_servo_demo")
    rospy.Subscriber('/ronex/general_io/12/state', GeneralIOState, callback)
    rospy.loginfo('RoNeX Pot Servo Demo Started!')
    rospy.spin()
```

2.2.21 The Code Explained

```
if __name__ == "__main__":

    rospy.init_node("ronex_pot_servo_demo")
    rospy.Subscriber('/ronex/general_io/12/state', GeneralIOState, callback)
    rospy.loginfo('RoNeX Pot Servo Demo Started!')
    rospy.spin()
```

In the main function we initialise our ROS node, then setup a subscriber to receive the position of the potentiometer on the state topic. Then an info message is printed to let the user know the program is running, then spin() is called to get the node up and running.

```
def callback(data):

    analogue = data.analogue
    pwmTopic = '/ronex/general_io/12/command/pwm/0'
    pub = rospy.Publisher(pwmTopic, PWM)
```

In the callback function the analogue data is retrieved from the message, and a publisher is created to send the required PWM command to the appropriate topic.

```
pwm = PWM()
pwm.pwm_period = 64000
pwm.pwm_on_time_0 = int(2.6042*analogue[0] + 1600)
pwm.pwm_on_time_1 = 0
pub.publish(pwm)
```

Here a message of format PWM is created, and filled with the required fields.

The calculations for converting the analogue value from the potentiometer into a corresponding PWM value for the servo can be found here: Analogue to PWM Configuration.

Note: The video shows a value of 1.5625 instead of 2.6042, this was mapping the 300° range of the potentiometer to the 180° range of the servo. In this case we directly map the bottom 180° of the potentiometer to the servo.

2.2.22 Running the Code

You can copy this straight into a new ROS package, be sure to make it executable using chmod +x if you want to run it using rosrun.

When you run the script, be sure that input_mode is set to false for the PWM signal channel, and that the publisher plugin in the GUI is not publishing.

2.2.23 Beginner Tutorials

1. **Read Analogue Data:** These examples how to read data from the analogue channels into your program: [Command Line | rqt_gui | Python | C++]
2. **Flash an LED using PWM:** These examples show you how to use a PWM module to to flash an LED connected to your GIO module: [Command Line | rqt_gui | Python | C++]
3. **Changing GIO Configuration:** These examples show how to dynamically change the configuration of your GIO module such as setting digital channels as inputs or outputs, from your program: [Command Line | rqt_gui | Python | C++]

2.2.24 Advanced Tutorials

4. Working with a joint based robot
5. Accessing the RoNeX data directly from a controller
6. Youtube Interfacing Video Code

This section will tell you everything you need to know to get started with a GIO module, including the interfaces provided and various ways to interact with the module through said interfaces.

Here is a link to the GIO datasheet.

2.3 Precursor: *General RoNeX Setup*

Before starting with the GIO module, you will need to run through the General RoNeX setup procedures in the sections listed on the General RoNeX Setup page.

2.4 *GIO Module System Overview*

Once you've set up your system and launched the driver, you can start to familiarise yourself with the GIO module.

First of all let's have a look through the interfaces the GIO module provides in the GIO Module System Overview.

2.5 *GIO Examples*

This section includes examples of how to interact with your GIO Module using the interfaces listed above.

CHAPTER 3

SPI Module

3.1 SPI Module Overview

This page describes some of the key parameters, services and topics you'll need to interact with when using the RoNeX SPI module. The information here builds on that related to the general RoNeX system which can be found here.

3.1.1 Data Flow

A priority when developing the RoNeX drivers is to ensure the ease of use of our product. For the SPI module, we developed a very generic driver that packs and unpacks the data into (and from) the proper etherCAT format so that it's easily accessible in the rest of the code.

By default a passthrough controller is loaded. This makes it possible to easily send / receive data to / from the SPI using a ROS service. Since it's a ROS service, you can access it from any of the usual languages in ROS: c++, python, command line.

Using that service is very convenient for a variety of use cases, but it will not be enough if you need to control exactly what's being sent to the SPI at each tick. For this you'll need to develop your own controller as explained in the Tutorials

3.1.2 Services

Any of these services can be called directly from the command line using rosservice call, or from a program as described in the Tutorials.

- `/ronex/spi/12/command/passthrough/[0..4]`

This service can be used to send and receive data from the SPI module.

3.1.3 Dynamic Reconfigure

Different configurations are available through the dynamic reconfigure interface for the SPI module.

Generic module configuration

- `command_type`: Type of the command to be sent (normal / config). This is set for the spi module as a whole.
- `pin_output_state_pre_DIO_0`: The pre state pin for DIO 0

- pin_output_state_post_DIO_0: The post state pin for DIO 0
- pin_output_state_pre_DIO_1: The pre state pin for DIO 1
- pin_output_state_post_DIO_1: The post state pin for DIO 1
- pin_output_state_pre_DIO_2: The pre state pin for DIO 2
- pin_output_state_post_DIO_2: The post state pin for DIO 2
- pin_output_state_pre_DIO_3: The pre state pin for DIO 3
- pin_output_state_post_DIO_3: The post state pin for DIO 3
- pin_output_state_pre_DIO_4: The pre state pin for DIO 4
- pin_output_state_post_DIO_4: The post state pin for DIO 4
- pin_output_state_pre_DIO_5: The pre state pin for DIO 5
- pin_output_state_post_DIO_5: The post state pin for DIO 5
- pin_output_state_pre_CS_0: The pre state pin for spi CS 0
- pin_output_state_post_CS_0: The post state pin for spi CS 0
- pin_output_state_pre_CS_1: The pre state pin for spi CS 1
- pin_output_state_post_CS_1: The post state pin for spi CS 1
- pin_output_state_pre_CS_2: The pre state pin for spi CS 2
- pin_output_state_post_CS_2: The post state pin for spi CS 2
- pin_output_state_pre_CS_3: The pre state pin for spi CS 3
- pin_output_state_post_CS_3: The post state pin for spi CS 3

Per SPI line configuration

The 0 in the following configurations can be replaced by 1,2 or 3 since there are 4 SPI lines per module.

- spi_mode_0: Select SPI Mode for SPI 0 - specify any mode of the spi_mode_enum.
- spi_0_input_trigger: The input trigger config - SPI[0]
- spi_0_mosi_somi: The MOSI SOMI pin config - SPI[0]
- spi_0_inter_byte_gap: The Inter Byte Gap - SPI[0]
- spi_0_clock_divider: The Clock Divider - SPI[0]

Values used for some of the parameters

- spi_mode_enum: SPI modes.

Mode	doc
NL_RE	Clock normally low, sample on rising edge
NL_FE	Clock normally low, sample on falling edge
NH_FE	Clock normally high, sample on falling edge
NH_RE	Clock normally high, sample on rising edge

- spi_input_trigger_enum: Input trigger configurations.

Mode	doc
SPI_CONFIG_INPUT_TRIGGER_NONE	SPI input trigger NONE
SPI_CONFIG_INPUT_TRIGGER_D0	SPI input trigger D0
SPI_CONFIG_INPUT_TRIGGER_D1	SPI input trigger D1
SPI_CONFIG_INPUT_TRIGGER_D2	SPI input trigger D2
SPI_CONFIG_INPUT_TRIGGER_D3	SPI input trigger D3
SPI_CONFIG_INPUT_TRIGGER_D4	SPI input trigger D4
SPI_CONFIG_INPUT_TRIGGER_D5	SPI input trigger D5

- `spi_mosi_somi_enum`: MOSI SOMI pin state.

Mode	doc
SPI_CONFIG_MOSI_SOMI_DIFFERENT_PIN	SPI MOSI SOMI different pin
SPI_CONFIG_MOSI_SOMI_SAME_PIN	SPI MOSI SOMI same pin

3.2 Tutorials

This section will tell you everything you need to know to get started with an SPI module, including the interfaces provided and various ways to interact with the module through said interfaces.

Here's a link to the SPI datasheet.

3.3 Precursor: *General RoNeX Setup*

Before starting with the SPI module, you will need to run through the General RoNeX setup procedures in the sections listed on the General RoNeX Setup page.

3.4 *SPI Module System Overview*

Once you've set up your system and launched the driver, you can start to familiarise yourself with the SPI module.

First of all let's have a look through the interfaces the SPI module provides in the SPI Module System Overview.

3.5 *SPI Tutorials*

This section includes examples of how to interact with your SPI Module.

CHAPTER 4

Special Use Cases

4.1 RoNeX on Windows with Matlab

This tutorial shows how to interact with RoNeX from MATLAB with minimal prior knowledge of Linux and ROS. RoNeX drivers run on Ubuntu within a VirtualBox virtual machine (A pre-configured VM image is available for download) so a native Linux install is not required. We use the MATLAB ROS support package under MATLAB R2013a or R2013b to send/receive data from RoNeX and process it in MATLAB.

For this tutorial we used Windows 8 64 bit install as the host system, although it should work on other versions of Windows without issues. It is running on a Lenovo Notebook computer, with a RoNeX stack connected to the Ethernet port, and a Wifi card connected to a network using DHCP.

If you experience problems during this tutorial, please let us know!

4.1.1 Installing VirtualBox + RoNeX Image

First download + install VirtualBox. The latest install file can be downloaded from: https://www.virtualbox.org/wiki/Downloads

Next install a Virtual Machine image with Ubuntu, ROS and the RoNeX drivers.

4.1.2 Configure VirtualBox

Once installed, open VirtualBox, go to File > Import Appliance, then find the Ubuntu ROS_RoNeX.ova system image you've just downloaded. You can adjust the amount of resources available to the VM here if you like, otherwise click import.

The host machine has both Wired and Wireless network cards configured to obtain an IP address automatically using DHCP. Once you have imported the virtual machine, you will need to open Settings > Network for that machine, and update the settings for adapter 1 (wired) and adapter 2 (wireless). By default the network adapter names correspond to those on our test machine, so you will need to select the correct names for your machine from the drop-down list. Ensure both adapters are attached to: "Bridged Adapter". You shouldn't need to adjust the advanced settings.

Ensure the machine is connected to your wifi network and has obtained an IP address. If you don't have a wireless or second wired card for a network connection, it is possible to set up static IPs for communication between MATLAB and VirtualBox, but that is beyond the scope of this tutorial.

4.1.3 Install MATLAB + ROS toolbox

To interact with ROS you will require MATLAB R2013a or R2013b. If you don't have MATLAB already, you can head over to the Mathworks website to find out how to get it:

http://www.mathworks.com/

Once MATLAB is set up, you will need to set up ROS support by downloading it from here:

http://www.mathworks.com/hardware-support/robot-operating-system.html

4.1.4 Configure ROS inside Virtual Machine

With your network adapters correctly configured in VirtualBox, once Ubuntu has booted, it should have obtained a unique IP on your network. You can check this IP by opening a terminal window (ctrl+alt+t) and typing:

```
ifconfig
```

Under adapter eth1 - inet addr: you should see the IP assigned to the virtual machine, make a note of it. Now in the same terminal we're going to set this IP in our ROS configuration. Open up the .bashrc config file by typing:

```
gedit ~/.bashrc
```

Scroll to the bottom of the file, and you should see the following lines:

```
export ROS_HOSTNAME=192.168.0.14
export ROS_MASTER_URI=http://192.168.0.14:11311
```

Change the IP address there to the one obtained from ifconfig. Be sure to leave the "http://" and ":11311" on the ROS_MASTER_URI.

4.1.5 Hardware Configuration

As previously mentioned, RoNeX is connected to the wired Ethernet port on the Laptop, which should now be set as eth0 in the virtual machine.

Connect one of your pre-made RoNeX GIO cables, to the analogue port on the GIO node, then connect 5V, GND and Channel 0 to your potentiometer, preferably on a solderless breadboard.

4.1.6 MATLAB Code

The MATLAB code below starts a RoNeX node and subscriber to receive RoNeX data.

The first section should be copied to a new MATLAB script in your workspace, named rostest.m

```
rosinit('http://192.168.0.14:11311')
sub = rossubscriber('/ronex/general_io/12/state', @my_function, 'Buffersize', 10))

i = 'a';
while i ~= 'q'
    i = input('q to quit', 's')
end

rosshutdown
clear;
```

This code should be copied into a file named my_function.m, in the same directory as rostest.m

```
function my_function(~, message)
    mydata = message.Analogue;
    disp (mydata(1));
end
```

4.1.7 Modifying the MATLAB Code

You will need to change the first line of the MATLAB script, to ensure the IP address listed for the ROS master, matches that we obtained from Ubuntu in the ROS configuration step.

```
rosinit('http://192.168.0.14:11311')
```

4.1.8 The Code Explained

The MATLAB code we are running is fairly simple, you can find a PDF manual with more information on the various functions in the MATLAB ROS Support package you downloaded. We'll start by looking at rostest.m:

```
rosinit('http://192.168.0.14:11311')
sub = rossubscriber('/ronex/general_io/12/state', @my_function, 'Buffersize', 10)
```

All of the ROS setup is done in these three lines. In the first a ROS node is defined (the user could choose an appropriate name), with the address of the ROS master.

Secondly a subscriber is defined to receive messages on the specified topic. The RoNeX message formats are included in the MATLAB ROS package by default, so we don't have to worry about custom messages. We set the callback function for the subscriber, i.e. whenever a valid message is received on the topic, this function is executed.

Finally we set a queue size of 10, this may need to be increased on slower machines so they don't miss messages.

```
i = 'a';
while i ~= 'q'
    i = input('q to quit', 's')
end
```

This code is to allow us to quit the program cleanly. It loops while waiting for keyboard input, if the received input character is a 'q', it will move on to shutdown the node.

```
rosshutdown
clear;
```

Once the quit message has been received, the node is shutdown and then everything is removed from the variable workspace using the clear command.

Next we will look at the my_function.m file:

```
function my_function(~, message)
    mydata = message.Analogue;
    disp (mydata(1));
end
```

This function is called every time a message is received on the corresponding topic, and is passed the message as an input argument. MATLAB allows to access the fields of the message but renames them to a camel case format instead of the lower case with underscores that is usual in ROS. The RoNeX state message has an analogue field, so we can access the Analogue field in the message (we could similarly access Digital).

Once we have the data, as the potentiometer is connected to channel 0, we are interested in the first value in the array, so we display this value then wait for the next message.

4.1.9 Execute Programs

First make sure ROS and the RoNeX driver are running in the virtual machine, as described in (see Launch driver).

Next run the MATLAB script (Either by right clicking on it in the current folder window and clicking run, typing the name of the file in the command window, or pressing f5 in the rostest.m editor window)

You should see the data from the potentiometer displayed in the command window, as you turn the pot, the values will reflect the angle of the shaft. Press q then enter when you are ready to quit the program.

4.2 RoNeX on Raspberry Pi

This is highly experimental!

4.2.1 Using pre-built image

You can download a pre-built and ready to use image, that can be copied to a 16GB (or bigger) SD card.

The following commands show a way to do it. sdx should be replaced with the device name that your system assigns to the SD card when you plug it in (e.g. using a card reader).

Bear in mind that this process will destroy any data on that SD card.

```
cd ~/Downloads
wget -O rPi_ronex.gz "https://doc-0c-2c-docs.googleusercontent.com/docs/securesc/ha0ro937gcuc7l7deff
gunzip rPi_ronex.gz
sudo dd if=~/Downloads/rPi_ronex of=/dev/sdx bs=1M
```

If the previous download link doesn't work, please try to download the image from the following link: rPi_ronex image.

User and password

This image has the default user and password for NOOBS install procedure:

```
user: pi
pwd: raspberry
```

4.2.2 Building step by step

If you don't want to use the pre-built image, you can install raspbian on your SD card, and install the necessary software for Ronex by following this step by step procedure.

References

The main install procedure has been taken from:

Setting up Hydro on RaspberryPi

It is reproduced here with some slight changes to have full step by step procedure.

Raspberry Pi hardware version

This procedure has been tested on a Raspberry Pi with 256 MB RAM.

Install Raspbian

- Start with NOOBS. There are several ways to do this, and plenty of tutorials on the internet.
- Install Raspbian from NOOBS.

Tweaks to the OS

On a 256 MB RAM board, we will need to maximise the available RAM and increase the swap size to succeed in the compilation of the necessary packages.

- Reduce GPU memory size to allow more ram:

```
sudo nano /boot/config.txt
```

then add or edit:

```
gpu_mem=16
```

- Change the size of the swap space in /etc/dphys-swapfile to 500MB (for compilation at least. It could be set back to 100 MB afterwards)

```
sudo nano /etc/dphys-swapfile
```

- It is also possible to overclock the arm processor:

```
sudo nano /boot/config.txt
```

then edit

```
arm_freq=800
```

All these changes require a reboot to take effect.

Install ROS and the necessary packages for Ronex

- Install repositories:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu raring main" > /etc/apt/sources.list.d/ros-l
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
sudo apt-get upgrade
```

- Install the dependencies:

```
sudo apt-get install python-pip
sudo pip install rosdistro
sudo pip install wstool
```

- Install setup tools:

```
cd ~/Downloads
wget https://pypi.python.org/packages/source/s/setuptools/setuptools-1.1.6.tar.gz
tar xvf setuptools-1.1.6.tar.gz
cd setuptools-1.1.6
sudo python setup.py install
```

- Install stdeb:

```
sudo apt-get install python-stdeb
```

- Install dependencies

```
sudo pip install rosdep
sudo pip install rosinstall-generator
sudo pip install wstool

pypi-install rospkg
sudo apt-get install python-rosdep python-rosinstall-generator build-essential
```

- Now continue from the Hydro install from source page. This downloads all the packages, and takes a couple hours.

```
mkdir ~/ros_catkin_ws
cd ~/ros_catkin_ws
rosinstall_generator ros_comm sr_ronex --rosdistro hydro --deps --wet-only > hydro-sr_ronex-wet.rosi
wstool init -j8 src hydro-sr_ronex-wet.rosinstall
sudo rosdep init
rosdep update
rosdep install  --from-paths src --ignore-src --rosdistro hydro -y --os=debian:wheezy
```

Now the rosdep fails :

```
Package sbcl is not available, but is referred to by another package.
This may mean that the package is missing, has been obsoleted, or
is only available from another source
However the following packages replace it:
  sbcl-source sbcl-doc

E: Package 'sbcl' has no installation candidate
ERROR: the following rosdeps failed to install
  apt: command [sudo apt-get install -y sbcl] failed
```

Apparently the roslisp package uses sbcl, which is not available for the pi, so we have to remove that.

```
cd src
wstool rm roslisp
rm -rf roslisp
cd ..
$
rosdep install  --from-paths src --ignore-src --rosdistro hydro -y --os=debian:wheezy
```

That worked! Now check that the ethercat_hardware package is at least in the version 1.8.6:

```
cd ~/ros_catkin_ws/src
wstool info pr2_ethercat_drivers/ethercat_hardware

If it's not (i.e it is in 1.8.5-0) then do:
wstool set pr2_ethercat_drivers/ethercat_hardware -v release/hydro/ethercat_hardware/1.8.6-0
wstool up pr2_ethercat_drivers/ethercat_hardware
```

- For a Raspberry Pi it is recommended to reduce the realtime loop frequency from the default 1 KHz to 250 Hz. An easy way to do it is to download and apply this patch:

```
cd ~/Downloads
wget -O reduced_loop_freq.patch "https://doc-0o-2c-docs.googleusercontent.com/docs/securesc/ha0ro937g
cd ~/ros_catkin_ws/src/pr2_ethercat
patch -p1 < ~/Downloads/reduced_loop_freq.patch
```

- Now it's time to try building it:

```
cd ~/ros_catkin_ws
./src/catkin/bin/catkin_make_isolated --install
```

Success!!!!

Make sure you reference the newly created install:

```
cd ~
echo "source ~/ros_catkin_ws/install_isolated/setup.bash" >> .bashrc
source .bashrc
```

This section explains how to use RoNeX in non standard ways, such as through a virtual machine, on an ARM device or through MATLAB.

4.3 *RoNeX on Windows with MATLAB*

This guide shows to use RoNeX in a virtual machine under Windows as well as how to interface with MATLAB.

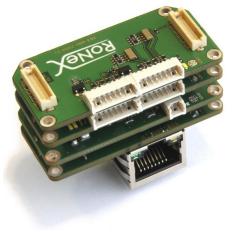
4.4 *RoNeX on RaspberryPi*

This guide explains how to install ROS and RoNeX drivers on a Raspberry Pi, for a compact control solution.

RoNeX is an industrial strength fieldbus for robots, that allows you to connect laptops, workstations or server farms directly to your hardware, in real time. To find out more, head over to the RoNeX Homepage.

This wiki contains RoNeX documentation. Please click on one of the links below to get started. If you can't find the information you are looking for, please let us know!

RoNeX user support, feedback and discussions in our Google Groups based forum RoNeX Sig.



CHAPTER 5

General RoNeX Setup

This section explains how to get started with a RoNeX system, including setting up your computer, launching the drivers and a run through overall layout of the system. This information is universal, regardless of which functional modules you are using.

CHAPTER 6

GIO Module Manual

This section will tell you everything you need to know to get started with a GIO module, including the interfaces provided and various ways to interact with the module through said interfaces.

CHAPTER 7

SPI Module Manual

This section will tell you everything you need to know to get started with an SPI module, including the interfaces provided and various ways to interact with the module through said interfaces.

CHAPTER 8

Special Use Cases

This section explains how to use RoNeX in non standard ways, such as through a virtual machine, on an ARM device or through MATLAB.

CHAPTER 9

Purchase RoNeX

If you don’t already have RoNeX hardware, you can head over to our Online Shop to buy it now!

CHAPTER 10

EtherCAT Conformant

Although there are plans to make RoNeX etherCAT conformant, it is not yet the case. This means that RoNeX doesn't work with programs like TwinCAT.