IPA CANopen User Manual

Version 0.01

Authors: Tobias Sing
Thiago de Freitas Oliveira Araujo
Eduard Herkel

Fraunhofer Institute for
Manufacturing Engineering and Automation

Stuttgart, Germany

Last modified on Tuesday 30th July, 2013

Contents

Chapter 1

Introduction

This manual describes the IPA CANopen library, a C++ framework for communicating with CANopen motor devices. Most users will use the ROS (Robot Operating System) wrapper provided as part of the package. However, the library itself is completely independent from ROS and therefore can also be used in other contexts.

The library and examples can be found in the following github repositories:

- `https://github.com/ipa320/ipa_canopen`
- `https://github.com/ipa320/ipa_canopen_tutorials`

In Section 2, a step-by-step guide to installing the library is given. Command-line tools for moving and referencing CANopen-enabled modules are described in Section 3. The control of CANopen modules from ROS is explained in Section 4. Finally, in Section 5, we show how to use and extend the library independently from ROS.

Chapter 2

Installation

At the moment, the first thing you have to do (if not done already) is to manually install the CAN device driver, as described in Section 2.2.

For the installation of the IPA CANopen library, there are two options:

- For usage from within ROS (Section 2.3)
- For using the C++ library without ROS (Section 2.4)

2.1 Prerequisites

You will need the following free tools, which are available for all operating systems:

- *CMake* (to manage the build process). It is pre-installed on many *nix operating systems. Otherwise, you need to install it first, e.g. in Ubuntu: `sudo apt-get install cmake`
- *git* (to download the sources from github). In Ubuntu, it can be installed with: `sudo apt-get install git`
- A C++ compiler with good support for the C++11 standard, e.g. *gcc* version 2.6 or higher (default in Ubuntu versions 11.04 or higher).

2.2 CAN device driver

Currently, the library has only been tested with the PCAN-USB CAN interface for USB from Peak System. It has been tested with version 7.6. The Linux user manual for the Peak interface is available at: `http://www.peak-system.com/fileadmin/media/linux/files/PCAN%20Driver%20for%20Linux_eng_7.1.pdf`. Briefly, to install the drivers under Linux, proceed as follows:

- Download and unpack the driver: `http://www.peak-system.com/fileadmin/media/linux/files/peak-linux-driver-7.6.tar.gz`.
- `cd peak-linux-driver-x.y`
- `make clean`
- Use the chardev driver: `make NET=NO`
  Note: The option `NET=no` is crucial!
- `sudo make install`
- `/sbin/modprobe pcan`

  If the Kernel is newer than 3.5 the following driver should be downloaded: `http://www.peak-system.com/fileadmin/media/linux/files/peak-linux-driver-7.7.tar.gz`
- Test that the driver is working:
  - `cat /proc/pcan` should look like this, especially `ndev` should be `NA`:

```
*------------ PEAK-System CAN interfaces (www.peak-system.com) ------------
*------------------------- Release_20120319_n (7.5.0) ----------------------
*---------------- [mod] [isa] [pci] [dng] [par] [usb] [pcc] -----------------
*--------------------- 1 interfaces @ major 248 found -----------------------
*n -type- ndev --base-- irq --btr- --read-- --write- --irqs-- -errors- status
32    usb -NA- ffffffff 255 0x001c 0000cc3f 0000edd1 00063ce1 00000005 0x0014
```

  - `./receivetest -f=/dev/pcan32` Turning the CAN device power on and off should trigger some CAN messages which should be shown on screen.
  - **Note: If you do not receive messages using this test, you still have issues with the device driver which need to be solved before proceeding further.**

2.3 IPA CANopen ROS package

- `git clone git://github.com/ipa320/ipa_canopen.git`
- `rosmake ros_canopen`
- Test if the installation was successful:
  - In one terminal: `roscore`
  - In another terminal: `rosrun ipa_canopen_ros canopen_ros`. This should give the following error message:

    ```
    Missing parameters on parameter server; shutting down node.
    ```
- Optionally, you can also install the tutorial examples:
  - `git clone git://github.com/ipa320/ipa_canopen_tutorials.git`
  - `rosmake ipa_canopen_tutorials`

2.4 ROS-indendent CANopen library

If you just want to use the C++ library independently from ROS, follow the steps described below:

2.4.1 Installation

- Go to a directory in which you want to create the source directory.
- `git clone git://github.com/ipa320/ipa_canopen.git`
- `cd ipa_canopen/ipa_canopen_core`
- Create a build directory and enter it: `mkdir build && cd build`
- Prepare the make files: `cmake ..`
- `make`
  - Optionally, you can make the installation available system-wide: `sudo make install`

- Test if the build was successful:
  - `cd tools`
  - `./homing`
  - This should give the output:

```
Arguments:
(1) device file
(2) CAN deviceID
Example: ./homing /dev/pcan32 12
```

Chapter 3

Command-line tools

The IPA Canopen library comes with two command-line tools for interacting with devices: *move_device* can be used to easily move a single device. *homing* tells the device to adjust its zero reference point to the current position.

Many devices do not have an absolute encoder. When disconnected from power, they rely on memory to store the current position of the device. Occasionally, these devices may become *dereferenced*, i.e. have a misadjusted zero position. In this case, manually moving the device to the true zero position using *move_device*, followed by a call to *homing* will readjust the device to the correct zero reference point.

Below, we will briefly describe both tools. When called without arguments, they will show their arguments along with a usage example.

3.1 The *homing* tool

This tool takes two arguments:

- The name of the CAN device file, e.g. /dev/pcan32
- The CAN device ID of the module to be homed, in decimal notation

When successfully evoking this command, you will usually either hear a clicking sound or see the device moving a bit.

3.2 The *move_device* tool

This tool takes five arguments:

- The name of the CAN device file, e.g. /dev/pcan32
- The CAN device ID of the module to be homed, in decimal notation
- The duration between two CANopen SYNC commands in milliseconds (if you do not know what this means, just use e.g. 10)
- The target velocity of the device in rad/sec
- The linear acceleration towards target velocity in rad/sec^2. Enter 0 to immediately request target velocity.

When first moving your device, it is recommended to start with low target velocities. Also note that some devices may enter an error or emergency state if the requested acceleration towards target velocity is too high.

3.3 The *get_error* tool

This tool takes two arguments:

- The name of the CAN device file, e.g. /dev/pcan32
- The CAN device ID of the module to be homed, in decimal notation

When successfully evoking this command, you will get the content of the errors registers from the device, naming the Manufacturer status register and the CANOpen standard errors register. This information is useful for debugging the status of the devices, and checking the source of problems when commanding the device.

Chapter 4

CANopen communication in ROS

In this chapter, we will show how to use the ipa_canopen_ros library for communicating with CANopen devices.

4.1 The tutorial examples

We have prepared a number of examples that cover the configuration steps which are necessary in order to use the library. These examples can be run via the launch file in their respective directory. As explained in Section 2.3, the examples can be installed as follows:

- `git clone git://github.com/ipa320/ipa_canopen_tutorials.git`
- `rosmake ipa_canopen_tutorials`

At the moment, there are 2×2 examples in the tutorials. The variations in these examples are:

- 1 CAN device vs. 3 CAN devices
- Direct sending of velocity commands to the ipa_canopen_ros node vs. using a trajectory controller to drive the device(s) to a desired position.

These examples should help you configuring the node for your own setup.

4.2 Getting started with ipa_canopen_ros

Before looking into the details, let's first try if everything works.

- `roscd ipa_canopen_tutorials/examples/1dof_simple/config`
- Edit the file `1dof_module.yaml` so that it contains a valid CAN id for your hardware (in decimal notation), and the correct device file (e.g. /dev/pcan32).
- In the same directory, edit the file `CANopen.yaml`, so that it contains the correct device file and the correct baudrate (e.g. 500K) to which the bus has been configured.
- `roslaunch 1dof_simple.launch`
- That's all! Now, after the ROS nodes have been launched, you should see your device moving slowly back and forth. If that's the case, your setup has been successful.

4.3 The canopen_ros node

In this section we will describe the `canopen_ros` node and the required ROS parameter settings, services, subscribers, and publishers. The node can be run by `rosrun ipa_canopen_ros canopen_ros`, but will usually be launched from a launch file.

To see tested configuration examples, please refer to the `config` and `urdf` subdirectories of the examples in the `ipa_canopen_tutorials` package, and to their corresponding launch files.

4.3.1 Parameters

In this section, we describe the necessary parameter settings.

- In the namespace in which the `canopen_ros` node is launched, the following parameters need to exist on the ROS parameter server:

  - `devices`: A list of CAN device files (`name`), its corresponding baud rates (`baudrate`), and SYNC intervals in msec (`sync_interval`). Example:

    ```
    rosparam get /canopen/devices
    - {baudrate: 500K, name: /dev/pcan32, sync_interval: 10}
    - {baudrate: 1M, name: /dev/pcan33, sync_interval: 5}
    ```

  - `chains`: A list of chain names (chain = group of devices). Each chain name must correspond to a namespace on the parameter server. Example:

    ```
    rosparam get /chains
    [arm1_controller, arm2_controller, tray_controller]
    ```

- For each of the chain names listed in parameter `chains` (cf. above), a namespace of that name must exist with the following parameters:

  - `joint_names`: A list of names/aliases for each joint in your chain.
  - `module_ids`: A list of corresponding CAN IDs (in decimal notation).
  - `devices`: A list of corresponding CAN device files (e.g. /dev/pcan32). The modules in a chain can be on different CAN buses.

4.3.2 Joint limits (urdf)

Note: Work in progress; joint limits and calibration are not yet processed and respected!

The ros_canopen node requires at least a rudimentary robot (urdf) model, from which it reads some information regarding the joints (cf. `http://www.ros.org/wiki/urdf/XML/joint`) from the `<joint>` tags. The parameters currently considered are:

- `<limit>` element **(mandatory)**: `velocity`, `lower`, `upper`
- `<calibration>` element (optional)

You can find example *xacro* files in the `urdf` subdirectories in the examples of `ipa_canopen_tutorials`.

4.3.3 Services, Subscribers, and Publishers

For each chain namespace listed in the `chains` parameter, the `ros_canopen` node subscribes to a required topic, publishes on a number of topics, and exposes the following services:

4.3.3.1 Services

- **init**: Service datatype `cob_srvs/Trigger` (`http://ros.org/doc/api/cob_srvs/html/srv/Trigger.html`). This opens the CAN bus connection(s) for the devices in the chain, if not already open, and puts the devices in operational mode.

- **recover**. Service datatype `cob_srvs/Trigger` (`http://ros.org/doc/api/cob_srvs/html/srv/Trigger.html`). If devices are in an emergency state, this will recover them to operational mode.

- **set_operation_mode**. Service datatype `cob_srvs/SetOperationMode` (`http://ros.org/doc/api/cob_srvs/html/srv/SetOperationMode.html`): not used at the moment.

4.3.3.2 Subscribers

- **command_vel**: Message datatype `brics_actuator/JointVelocities` (`http://www.ros.org/doc/api/brics_actuator/html/msg/JointVelocities.html`). Essentially a vector of desired velocities for each module in a chain.

4.3.3.3 Publishers

- **state**: Message datatype: `pr2_controllers_msgs/JointTrajectoryControllerState` (`http://www.ros.org/doc/api/pr2_controllers_msgs/html/msg/JointTrajectoryControllerState.html`). Vectors of desired and actual positions and velocities of all devices in a chain.

- **/joint_states**: Message datatype `sensor_msgs/JointState` (`http://www.ros.org/doc/api/sensor_msgs/html/msg/JointState.html`). Publisher

in the global namespace, taken up e.g. by the `robot_state_publisher` (`http://www.ros.org/wiki/robot_state_publisher`).

4.4 Driving the Schunk LWA4P arm

The Schunk LWA 4P robotic arm can be driven by CANopen commands. You can get the necessary configuration files by cloning the following two repositories:

- `git clone git@github.com:ipa320/schunk_robots.git`
- `git clone git@github.com:ipa320/schunk_modular_robotics.git`

To launch the CANopen driver, together with a trajectory controller:
`roslaunch schunk_bringup lwa4p_solo.launch`

To launch the Powerball arm in Gazebo simulation:
`roslaunch schunk_bringup_sim lwa4p.launch`

This will allow you to control the arm by via JointTrajectoryFollowActions. To test this, you can move to a small number of example configurations from a graphical command GUI:
`roslaunch schunk_bringup dashboard_lwa4p.launch`

4.4.1 Inverse kinematics

Work in progress

Chapter 5

Extending the CANopen library

Work in progress.

Chapter 6

Troubleshooting

For troubleshooting the device, the current recommendation is to run the *get_error tool to check the content of the error registers.*