canopen / ros_canopen

User Manual

Preliminary version, work in progress

Tobias Sing

Fraunhofer Institute for
Manufacturing Engineering and Automation

Stuttgart, Germany

Last modified on Wednesday 10th October, 2012

Contents

Chapter 1

Introduction

1.1 CANopen: A short introduction

In this section, we only briefly summarize the standard so that the rest of the manual can be understood more easily.

CANopen is a standard with two purposes:

- A high-level communication protocol. It specifies the format and mode of messages exchanged, but not the lower levels of physical transport. Most frequently, the physical transport is via the CAN field bus, although any other mean of transportation (e.g. EtherCAT) could also be used to implement the CANopen communication protocol.
- Several manufacturer-independent device profiles, e.g. for motors or sensors. This allows for example to use the same set of commands to communicate with motors from different manufacturers.

1.1.1 CANopen functions: The object dictionary (indices.csv)

ALL CANopen functions are entries in the *object dictionary*. These entries are identified by a 16-bit *index* and an 8-bit *subindex*. In the IPA CANopen library, the objects in the object dictionary are collected in a single space-delimited file `indices.csv` in the `driver` directory (it gets copied into `~/.canopen` by `sudo`

`make install`. Each line in the file represents one object from the standard. Currently, only a minimal set of the standard is collected in this file. Here is an excerpt:

```
controlword 0x6040 0x0 2 rw
statusword 0x6041 0x0 2 ro
modes_of_operation 0x6060 0x0 1 wo
modes_of_operation_display 0x6061 0x0 1 ro
torque_actual_value 0x6077 0x0 2 rw
position_actual_value 0x6064 0x0 4 rw
```

Each entry has five columns:

- *Alias*: how the object is called in the standard
- *Index*: The 16-bit main address (index) which identifies the object in the standard and in actual CANopen messages.
- *Subindex*: The 8-bit subindex which is used to access specific subfunctions for some objects.
- *Length*: The length in bytes (1-4) for the data part of an object. For "read-only" ("ro" in column 5) objects this is the length of data returned in a reply from the device, whereas for "read-write" objects ("rw" in column 5) it is the length of data that must be submitted when a message of that particular index/subindex type is sent to a device.
- *Mode*: Some objects are read-only (e.g. statusword), whereas others are read-write (e.g. controlword). This is specified in column 5 for each object.

To make available additional standard functionality within the API, you can simply add lines to the `indices.csv` file (cf. Chapter 5).

1.1.2 CANopen constants (constants.csv)

For many entries of the object dictionary, the CANopen standard defines constants with a specific functionality. For example, several bits in the data part of the *controlword* are used to trigger specific state machine transitions in motor (402

standard) devices. These constants are all collected in the file `constants.csv`, for example:

```
controlword disable_voltage 0xFFFF 0x1
controlword sm_shutdown 0xFFFF 0x6
controlword sm_switch_on 0xFFFF 0x7
controlword sm_enable_operation 0xFFFF 0xF
```

WARNING!!!! The value of a constant (last column) will always be interpreted in hexadecimal notation!!!

Each entry in this file has four columns:

- *Alias of object*: This is the name of the object for which the constant is defined.
- *Alias of constants*: This is the name of the constant in the standard.
- *Bit-mask*: Bits that are not set must also be evaluated. For that reason, data must be masked accordingly for evaluation.
- *Value*: The value that defines the constants (relative to the bit mask).

1.1.3 CANopen message types

In this library, all communcation is by objects of the class `canopen::Message` (`canopenmsg.h`).

The CANopen messages types relevant when using this library are:

- *NMT messages*: em NMT stands for *Network Management protocol*. NMT messages are sent out to the devices in order to invoke state transitions in the device-internal communication (301 standard) state machines. NMT messages, as implemented currently, apply simultaneously to all devices on a bus. All devices can be set to *operational* state (according to 301 state machine) using the function `canopen::initNMT` (`canopen_highlevel.h`). Individual NMT commands can be sent using the function `canopen::sendNMT` (`canopenmsg.h`), e.g. `sendNMT(‘‘stop_remote_node’’)`. The string argument is any constant alias defined in the file `constants.csv` (c.f. 1.1.2,

additional constants defined in the standard can be added by the user if necessary).

- *SDO messages*: *SDO* stands for *Service Data Object* protocol. Each SDO message is one object from the object dictionary (cf. **??**). SDOs are used to set and read values from devices. Typically, SDO communication is used for device initialization and setting specific communication or operation-mode parameters. Actual motion commands (e.g. positions, velocities) are usually not sent by SDOs, but rather by PDOs (cf. below). This is because SDO communication is always confirmed by the device, i.e. the device always send a reply back in response to an SDO message it receives from the master. For the high-frequency communication when sending motion commands this would be too much overhead. **In this library, SDO calls can be performed as function calls and the return value of these calls is the SDO reply received from the device**. This is implemented in the following way: whenever an SDO is sent out, it gets stored into the hash-table `canopen::pendinSDOreplies`. Once a reply has been received from the device, the function call returns with the reply as return value. SDOs can be sent using the `sendSDO()` function (`canopenmsg.h`), e.g.

  ```
  sendSDO(deviceID, "controlword", "sm_shutdown");
  Message* reply = sendSDO(deviceID, "statusword");
  ```

  In `canopen_highlevel.h`, higher-level functionality that consists of sequences of SDO calls is collected. For example, the function `canopen::homing()` puts a device into homing mode, the starts homing, then blocks while the drive is moving, and finally checks whether the drive reports itself as references, returning the result ina boolean value. All these commands and checks are performed by SDO communication.

- *PDO messages*: *PDO* stands for *Process Data Object*. Unlike SDO messages, PDO messages are unconfirmed (i.e. one-way) communication. PDOs sent from the devices to the master are referred to as tPDOs (transmit-PDOs), whereas PDOs sent from the master to the devices are referred to as rPDOs (receive-PDOs). PDOs are crucial for controlling devices and therefore are described below in more detail (cf. Section **??**).

- *SYNC messages*: SYNC messages are used to coordinate the processing of rPDOs and transfer of tPDOs by the devices. They ensure that different devices on the same bus perform the operations in a synchronized

way. Among several forms of PDO communication allowed by the standard, here only the SYNC-synchronized form is implemented so far. The SYNC message can be sent by `canopen::sendSync()` (`canopen_highlevel.h`). When using a *master* thread to coordinate the sending and receiving of messages, the master automatically takes care of things such as sending SYNC messages at the right times.

1.1.4 Controlling device motion with PDOs (Process Data Objects) - (PDOs.csv)

The data transmitted in a PDO consists of a collection of objects from the object dictionary. A device usually has some default PDOs, but a user can also specify custom PDOs. In this API, PDOs can be easily specified in the file `PDOs.csv`. Each row defines on PDO. First, the alias of a PDO is defined, e.g. `schunk_default_tPDO`. Then, the ID ("cobID") which identifies the message on the CAN bus is specified (e.g. 0x200). Then a sequence of aliases defined in the object dictionary are given, e.g. `statusword torque_actual_value position_actual_value`. Together, the transmitted data has to be 8 bytes (the lengths can be seen from the `length` column in *indices.csv*).

The specification of PDOs in the file PDOs only makes it possible within the API to easily generate and parse such messages. New user-defined PDOs also need to be declared to the devices using the function `canopen::definePDO` (not implemented yet).

Example definitions from `PDOs.csv`:

```
schunk_default_rPDO 0x200 controlword notused16 interpolation_data_record:ip_data_position
schunk_default_tPDO 0x180 statusword torque_actual_value position_actual_value
schunk_debug_rPDO 0x480 notused64
```

Chapter 2

Installation

2.1 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 7.5. The Linux user manual 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-7.5.tar.gz`.
- `cd peak-linux-driver-x.y`
- `make clean`
- Use the chardev driver: `make NET=NO`
- `sudo make install`
- `/sbin/modprobe pcan`
- 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.

2.2 ROS-indendent CANopen library

2.2.1 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).

2.2.2 Installation

- Go to a directory in which you want to create the *canopen* source directory.
- `git clone git://github.com/ipa-tys/canopen.git`
- `cd canopen`
- Now create a directory for building the code: `mkdir build`
- `cd build`
- Prepare the make files: `cmake ..`
- `make`

- (Optionally:) `sudo make install`
- Test if the build was successful:
  - `cd examples`
  - `./homing`
  - This should give the output:

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

2.3 IPA CANopen ROS package

Currently, this requires that you first perform the two installation steps (PCAN driver, ROS-independent CANopen library) above manually. Then:

- `git clone git://github.com/ipa-tys/ros_canopen.git`
- `rosmake ros_canopen`. Make sure dependencies are installed (e.g. package cob_srvs from stack cob_common).
- Test if the installation was successful:
  - In one terminal: `roscore`
  - In another terminal: `rosrun ros_canopen ros_canopenmasternode`. This should give the output:

```
File not found. Please provide a description file as command line argument
```

Chapter 3

Using the ROS-independent library

The library can be used on two different levels:

- By directly writing CANopen messages to the bus using the API functions provide in `canopen_highlevel.h`.

- On a higher level, there is also an implementation of a master thread with functionality to manage device objects and object groups.

We first describe the use of API functions to write directly to the CAN bus. This chapter assumes that you have completely all the installation steps described in Chapter **??**, including the building of the examples in the `demo` folder.

3.1 Using API functions to directly write to the CAN bus

The whole library lives within a dedicated namespace, `canopen`. Therefore, you will often see this as a prefix, e.g. `canopen::openConnection`.

3.1.1 Getting started: Communicating with a single device (demo/single_device.cpp)

This example shows how to communicate with a single device. If you invoke the script with the CAN ID (in decimal representation) of your device as a command-line parameter, e.g. `./single_device 12`, you should first see your item moving around until it is referenced (homing mode). Then the device will start moving in one direction in interpolated position (IP) mode. To understand how to use the library let's examine the code of the example[1].

```
#include <canopen_highlevel.h>
```

This is the header file to include if you just want to use API functions to directly write to the CAN bus, without using a master thread to manage device objects and object groups.

```
if (!canopen::openConnection("/dev/pcan32"))
```

This is always the first command to invoke when using the CANopen library. It attempts to open the device connection. It returns `true` in case of success and `false` otherwise.

```
canopen::initListenerThread();
```

Even in the direct communication without a *master thread*, there needs to be a separate thread to listen for incoming messages from the CAN bus. Without a master, this thread only performs the following action with the two types of messages coming in from the devices:

- *SDOs*: An SDOs coming in from a device is always a response to an SDO sent out from the master to a device. In this library, the user is shielded from this asynchronous communication. To the user, SDO communication simply appears as function calls with the return value of the function being the SDO reply coming from the device in response to the SDO sent out from the master.

[1] In this manual, only the relevant parts of the examples are shown. Go to the `demo` folder to see the full example code.

- *PDOs*: PDOs coming in from the devices (tPDOs) are simply taken by the listener thread and put into the queue `canopen::incomingPDOs`. When the master thread is used, it takes care of distributing the PDOs to the device objects and of interpreting them. Without using a master, users are responsible by themselves to take the incoming PDOs out of the queue and to interpret (or discard) them. Without any user action, the `canopen::incomingPDO` queue will grow indefinitely.

```
canopen::initNMT();
```

This puts the communication (301 standard) state machines of all devices on the bus into operational mode.

```
canopen::initNMT();

if (!canopen::initDevice(deviceID))
```

This puts the motor (402 standard) state machine of a device into operational mode and return true if the device reports itself as operational afterwards, and false otherwise.

```
if (!canopen::homing(deviceID))
```

This triggers a device to reference itself (homing). The function call blocks until the drive has stopped moving and then returns true if the device reports itself as referenced and false otherwise.

```
if (!canopen::enableIPmode(deviceID))
```

This makes a device ready to perform motion in the interpolated-position (IP) mode by receiving position commands via PDOs. The function call return *false* if the device could not be put into IP mode (this could be for example due to the device not supporting this drive mode at all).

```
canopen::sendPos(deviceID, pos);
canopen::sendSync(10);
```

When a device is in IP mode, it can be moved by alternating between sending `canopen::sendPos` and `canopen::sendSync` commands. If a master thread is used, the user does not have to take care of sending SYNC commands.

```
canopen::enableBreak(deviceID);
canopen::shutdownDevice(deviceID);
canopen::closeConnection();
```

These commands are used to put the motor state machine from operation into ready_to_switch_on (i.e. enable the motor brake), to shut down the communication (301) state machine, and to close the device connection, respectively.

3.1.2 Getting started: Communicating with multiple devices simultaneously (demo/multiple_devices.cpp)

This example shows how to communicate with a group ("chain") of devices, e.g. a robot arm such as the *Schunk Powerball* arm. If you invoke the script with the CAN IDs (in decimal representation) of your devices as a command-line parameter, e.g. `./multiple_devices 3 4 5 6 7 8`, you should first see all devices of the arm referencing themselves. Afterwards, you should see the arm performing some random motion.

If you compare `demo/single_device` and `demo/multiple_devices`, you will see the use of the API commands is completely identical in both cases. Of course, the API commands can also be used in this way to communicate simultaneously, for example, with an arm and a mobile base.

3.2 CANopen with a master thread

The *master thread* implemented in `canopenmaster.cpp` adds on top of the functionality seen in the previous examples. As has been seen above, its use is optional and the user is free to rely only on the message sending and parsing functionality implemented in the API.

The pre-implemented master thread can also be easily adapted or extended to accomodate other use cases (cf. Chapter 5).

The pre-implemented master also provides callback functions which are completely framework(e.g.ROS)-independent. These callbacks are intended to be wrapped easily for inter-process communication, as shown in the *ROS_canopen* package.

In the framework-independent examples of this section, we simply call the

3.2.1 canopenmaster_single_device example

In the file `demo/canopenmaster_single_device`, we show a very simple example of how to use the pre-implemented master thread. To show its use in a framework-independent way, we invoke the callbacks from a separate thread in the examples.

```
#include <canopenmaster.h>
```

This is where the master thread functionality comes from.

```
canopen::using_master_thread = true;
```

When using a master, you must set this namespace-global variable to *true*. Then any commands to send CAN messages are not sent directly to the bus, but rather are put on a sending queue which is controlled by the master. The master also has objects for all CAN devices and groups (chains) of CAN devices which have to be coordinated, e.g. a robot arm. The master will also take care of incoming PDOs from the devices and dispatch them to the device objects where they are used to update device object member variables (e.g. most recent position etc.) which can be queried at any time.

```
canopen::initChainMap("/home/tys/git/other/canopen/demo/single_device.csv");
```

A crucial part of the master is that it has an object for every device and for every group of devices (chain). The total of devices and chains is described in a small text file with one row per group (chain) of devices. Our minimalistic robot in this example consists only of a single device within a single chain. The config file partsed with the command above looks like this:

```
chain1 joint1 /dev/pcan32 12
```

First entry in the row is the chain name. All other entries are triples (devicename devicebus deviceCANid). The command `canopen::initChainMap` takes this specification and creates a map (hash table), called `canopen::chainMap` that links chain names to (pointers to) ojects of class `canopen::Chain` (`chain.h`).

```
if (!canopen::openConnection("/dev/pcan32")) {
  std::cout << "Cannot open CAN device; aborting." << std::endl;
  return -1;
}
canopen::initNMT();
canopen::initListenerThread();
```

These commands are already known from the direct API call examples above and are used in exactly the same way here.

```
canopen::initIncomingPDOProcessorThread();
```

Remember that without a master, incoming PDOs are simply stored in the queue `incomingPDOs`. Using a *master* thread implies having explicit object representations of devices and device groups (chains). These objects are created using the `canopen::initChainMap` command mentioned above. The command `initIncomingPDOProcessorThread` launches a thread that processes messages from this queue: based on the PDO cobID the PDOProcessor triggers a member function `canopen::Chain::updateStatusWithIncomingPDO` in the *canopen::Chain* object to which the device from which the PDO was sent belongs. This function can check if any object (index/subindex) from the object dictionary is contained in that PDO which it can use to update some of its status member variables. It then triggers a member function `canopen::Device::updateStatusWithIncomingPDO` of the object representing the device from which the PDO was sent. Again, this function can check for the presence of specific objects in the PDO using the member function `Message::find_component` (**todo**!) and update its status member variables. Currently, the only check performed is for the presence of the object `position_actual_value` (index 0x6064; mapped in the Schunk default

tPDO), which if present in a PDO is used to update the member variable `current_position_`. Other checks and updates can be performed similarly and incorporated into the function `Device::updateStatusWithIncomingPDO` to extend the functionality.

```
canopen::initMasterThread();

  while (true)
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
```

Chapter 4

CANopen with ROS

In this chapter, we describe the use of the CANopen library with ROS.

4.0.2 Getting started: Communicating with a single device

```
roscore
roslaunch
roslaunch ros_canopen single_device.launch
```

The library can be used on two different levels:

- By directly writing CANopen messages to the bus using the API functions provide in `canopen_highlevel.h`.
- On a higher level, there is also an implementation of a master thread with functionality to manage device objects and object groups.

We first describe the use of API functions to write directly to the CAN bus. This chapter assumes that you have completely all the installation steps described in Chapter **??**, including the building of the examples in the `demo` folder.

4.1 Using API functions to directly write to the CAN bus

The whole library lives within a dedicated namespace, `canopen`. Therefore, you will often see this as a prefix, e.g. `canopen::openConnection`.

4.1.1 Getting started: Communicating with a single device (demo/single_device.cpp)

This example shows how to communicate with a single device. If you invoke the script with the CAN ID (in decimal representation) of your device as a command-line parameter, e.g. `./single_device 12`, you should first see your item moving around until it is referenced (homing mode). Then the device will start moving in one direction in interpolated position (IP) mode. To understand how to use the library let's examine the code of the example[1].

```
#include <canopen_highlevel.h>
```

This is the header file to include if you just want to use API functions to directly write to the CAN bus, without using a master thread to manage device objects and object groups.

```
if (!canopen::openConnection("/dev/pcan32"))
```

This is always the first command to invoke when using the CANopen library. It attempts to open the device connection. It returns `true` in case of success and `false` otherwise.

```
canopen::initListenerThread();
```

Even in the direct communication without a *master thread*, there needs to be a separate thread to listen for incoming messages from the CAN bus. Without a master, this thread only performs the following action with the two types of messages coming in from the devices:

[1] In this manual, only the relevant parts of the examples are shown. Go to the `demo` folder to see the full example code.

- *SDOs*: An SDOs coming in from a device is always a response to an SDO sent out from the master to a device. In this library, the user is shielded from this asynchronous communication. To the user, SDO communication simply appears as function calls with the return value of the function being the SDO reply coming from the device in response to the SDO sent out from the master.

- *PDOs*: PDOs coming in from the devices (tPDOs) are simply taken by the listener thread and put into the queue `canopen::incomingPDOs`. When the master thread is used, it takes care of distributing the PDOs to the device objects and of interpreting them. Without using a master, users are responsible by themselves to take the incoming PDOs out of the queue and to interpret (or discard) them. Without any user action, the `canopen::incomingPDO` queue will grow indefinitely.

```
canopen::initNMT();
```

This puts the communication (301 standard) state machines of all devices on the bus into operational mode.

```
canopen::initNMT();
```

```
if (!canopen::initDevice(deviceID))
```

This puts the motor (402 standard) state machine of a device into operational mode and return true if the device reports itself as operational afterwards, and false otherwise.

```
if (!canopen::homing(deviceID))
```

This triggers a device to reference itself (homing). The function call blocks until the drive has stopped moving and then returns true if the device reports itself as referenced and false otherwise.

```
if (!canopen::enableIPmode(deviceID))
```

This makes a device ready to perform motion in the interpolated-position (IP) mode by receiving position commands via PDOs. The function call return *false* if the device could not be put into IP mode (this could be for example due to the device not supporting this drive mode at all).

```
canopen::sendPos(deviceID, pos);
canopen::sendSync(10);
```

When a device is in IP mode, it can be moved by alternating between sending `canopen::sendPos` and `canopen::sendSync` commands. If a master thread is used, the user does not have to take care of sending SYNC commands.

```
canopen::enableBreak(deviceID);
canopen::shutdownDevice(deviceID);
canopen::closeConnection();
```

These commands are used to put the motor state machine from operation into ready_to_switch_on (i.e. enable the motor brake), to shut down the communication (301) state machine, and to close the device connection, respectively.

4.1.2 Getting started: Communicating with multiple devices simultaneously (demo/multiple_devices.cpp)

This example shows how to communicate with a group ("chain") of devices, e.g. a robot arm such as the *Schunk Powerball* arm. If you invoke the script with the CAN IDs (in decimal representation) of your devices as a command-line parameter, e.g. `./multiple_devices 3 4 5 6 7 8`, you should first see all devices of the arm referencing themselves. Afterwards, you should see the arm performing some random motion.

If you compare `demo/single_device` and `demo/multiple_devices`, you will see the use of the API commands is completely identical in both cases. Of course, the API commands can also be used in this way to communicate simultaneously, for example, with an arm and a mobile base.

4.2 CANopen with a master thread

The *master thread* implemented in `canopenmaster.cpp` adds on top of the functionality seen in the previous examples. As has been seen above, its use is optional and the user is free to rely only on the message sending and parsing functionality implemented in the API.

The pre-implemented master thread can also be easily adapted or extended to accomodate other use cases (cf. Chapter 5).

The pre-implemented master also provides callback functions which are completely framework(e.g.ROS)-independent. These callbacks are intended to be wrapped easily for inter-process communication, as shown in the *ROS_canopen* package.

In the framework-independent examples of this section, we simply call the

4.2.1 canopenmaster_single_device example

In the file `demo/canopenmaster_single_device`, we show a very simple example of how to use the pre-implemented master thread. To show its use in a framework-independent way, we invoke the callbacks from a separate thread in the examples.

```
#include <canopenmaster.h>
```

This is where the master thread functionality comes from.

```
canopen::using_master_thread = true;
```

When using a master, you must set this namespace-global variable to *true*. Then any commands to send CAN messages are not sent directly to the bus, but rather are put on a sending queue which is controlled by the master. The master also has objects for all CAN devices and groups (chains) of CAN devices which have to be coordinated, e.g. a robot arm. The master will also take care of incoming PDOs from the devices and dispatch them to the device objects where they are

used to update device object member variables (e.g. most recent position etc.) which can be queried at any time.

```
canopen::initChainMap("/home/tys/git/other/canopen/demo/single_device.csv");
```

A crucial part of the master is that it has an object for every device and for every group of devices (chain). The total of devices and chains is described in a small text file with one row per group (chain) of devices. Our minimalistic robot in this example consists only of a single device within a single chain. The config file partsed with the command above looks like this:

```
chain1 joint1 /dev/pcan32 12
```

First entry in the row is the chain name. All other entries are triples (devicename devicebus deviceCANid). The command `canopen::initChainMap` takes this specification and creates a map (hash table), called `canopen::chainMap` that links chain names to (pointers to) ojects of class `canopen::Chain` (`chain.h`).

```
if (!canopen::openConnection("/dev/pcan32")) {
  std::cout << "Cannot open CAN device; aborting." << std::endl;
  return -1;
}
canopen::initNMT();
canopen::initListenerThread();
```

These commands are already known from the direct API call examples above and are used in exactly the same way here.

```
canopen::initIncomingPDOProcessorThread();
```

Remember that without a master, incoming PDOs are simply stored in the queue `incomingPDOs`. Using a *master* thread implies having explicit object representations of devices and device groups (chains). These objects are created using the `canopen::initChainMap` command mentioned above. The command `initIncomingPDOProcessorThread` launches a thread that processes messages from this queue: based on the PDO cobID the PDOProcessor triggers a member function `canopen::Chain::updateStatusWithIncomingPDO` in

the *canopen::Chain* object to which the device from which the PDO was sent belongs. This function can check if any object (index/subindex) from the object dictionary is contained in that PDO which it can use to update some of its status member variables. It then triggers a member function `canopen::Device::updateStatusWithIncomingPDO` of the object representing the device from which the PDO was sent. Again, this function can check for the presence of specific objects in the PDO using the member function `Message::find_component` (**todo**!) and update its status member variables. Currently, the only check performed is for the presence of the object `position_actual_value` (index 0x6064; mapped in the Schunk default tPDO), which if present in a PDO is used to update the member variable `current_position_`. Other checks and updates can be performed similarly and incorporated into the function `Device::updateStatusWithIncomingPDO` to extend the functionality.

```
canopen::initMasterThread();

  while (true)
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
```

Chapter 5

Extending the CANopen library