Contents

1 Network and nodes 3
2 Object Dictionary 11
3 Network management (NMT) 17
4 Service Data Object (SDO) 19
5 Process Data Object (PDO) 27
6 Synchronization Object (SYNC) 33
7 Emergency Object (EMCY) 35
8 Time Stamp Object (TIME) 37
9 Layer Setting Services (LSS) 39
10 Integration with existing code 43
11 Device profiles 45
Index 47
This package provides support for interacting with a network of CANopen nodes.

**Note:** Most of the documentation here is directly stolen from the CANopen Wikipedia page.
This documentation is a work in progress. Feedback and revisions are most welcome!

CANopen is a communication protocol and device profile specification for embedded systems used in automation. In terms of the OSI model, CANopen implements the layers above and including the network layer. The CANopen standard consists of an addressing scheme, several small communication protocols and an application layer defined by a device profile. The communication protocols have support for network management, device monitoring and communication between nodes, including a simple transport layer for message segmentation/desegmentation.

Easiest way to install is to use **pip**:

```
$ pip install canopen
```
Network and nodes

The `canopen.Network` represents a collection of nodes connected to the same CAN bus. This handles the sending and receiving of messages and dispatches messages to the nodes it knows about.

Each node is represented using the `canopen.RemoteNode` or `canopen.LocalNode` class. It is usually associated with an object dictionary and each service has its own attribute owned by this node.

### 1.1 Examples

Create one network per CAN bus:

```python
import canopen

network = canopen.Network()
```

By default this library uses `python-can` for the actual communication. See its documentation for specifics on how to configure your specific interface.

Call the `connect()` method to start the communication, optionally providing arguments passed to a the `can.BusABC` constructor:

```python
network.connect(channel='can0', bustype='socketcan')
# network.connect(bustype='kvaser', channel=0, bitrate=250000)
# network.connect(bustype='pcan', channel='PCAN_USBBUS1', bitrate=250000)
# network.connect(bustype='ixxat', channel=0, bitrate=250000)
# network.connect(bustype='nican', channel='CAN0', bitrate=250000)
```

Add nodes to the network using the `add_node()` method:

```python
node = network.add_node(6, '/path/to/object_dictionary.eds')
local_node = canopen.LocalNode(1, '/path/to/master_dictionary.eds')
network.add_node(local_node)
```
Nodes can also be accessed using the `Network` object as a Python dictionary:

```python
for node_id in network:
    print(network[node_id])
```

To automatically detect which nodes are present on the network, there is the `scanner` attribute available for this purpose:

```python
# This will attempt to read an SDO from nodes 1 - 127
network.scanner.search()
# We may need to wait a short while here to allow all nodes to respond
time.sleep(0.05)
for node_id in network.scanner.nodes:
    print("Found node \$d!" % node_id)
```

Finally, make sure to disconnect after you are done:

```python
network.disconnect()
```

## 1.2 API

```python
class canopen.Network(bus=None)
    Representation of one CAN bus containing one or more nodes.

    Parameters
    ----------
    bus : can.BusABC
        A python-can bus instance to re-use.

    nmt
        The broadcast `canopen.nmt.NmtMaster` which will affect all nodes.

    sync
        The `canopen.sync.SyncProducer` for this network.

    time
        The `canopen.timestamp.TimeProducer` for this network.

    network[node_id]
        Return the `canopen.RemoteNode` or `canopen.LocalNode` for the specified node ID.

    iter(network)
        Return an iterator over the handled node IDs.

    node_id in network
        Return True if the node ID exists is handled by this network.

    del network[node_id]
        Delete the node ID from the network.

    values()
        Return a list of `canopen.RemoteNode` or `canopen.LocalNode` handled by this network.

    add_node(node, object_dictionary=None, upload_eds=False)
        Add a remote node to the network.

        Parameters
        ----------
        node
            Can be either an integer representing the node ID, a `canopen.RemoteNode` or `canopen.LocalNode` object.

        object_dictionary
            Can be either a string for specifying the path to an Object Dictionary file or a `canopen.ObjectDictionary` object.
```
• **upload_eds** *(bool)* – Set True if EDS file should be uploaded from 0x1021.

**Returns**  The Node object that was added.

**Return type** *canopen.RemoteNode*

```python
def canopen.Documentation, Release 1.0.0

- upload_eds (bool) – Set True if EDS file should be uploaded from 0x1021.

Returns  The Node object that was added.

Return type  canopen.RemoteNode

bus = None
A python-can can.BusABC instance which is set after canopen.Network.connect() is called

check ()
Check that no fatal error has occurred in the receiving thread.
If an exception caused the thread to terminate, that exception will be raised.

connect (*args, **kwargs)
Connect to CAN bus using python-can.
Arguments are passed directly to can.BusABC. Typically these may include:

Parameters
- **channel** – Backend specific channel for the CAN interface.
- **bustype** *(str)* – Name of the interface. See python-can manual for full list of supported interfaces.
- **bitrate** *(int)* – Bitrate in bit/s.

Raises  can.CanError – When connection fails.

create_node (node, object_dictionary=None)
Create a local node in the network.

Parameters
- **node** – An integer representing the node ID.
- **object_dictionary** – Can be either a string for specifying the path to an Object Dictionary file or a canopen.ObjectDictionary object.

Returns  The Node object that was added.

Return type  canopen.LocalNode

disconnect ()
Disconnect from the CAN bus.
Must be overridden in a subclass if a custom interface is used.

listeners = None
List of can.Listner objects. Includes at least MessageListener.

notify (can_id, data, timestamp)
Feed incoming message to this library.
If a custom interface is used, this function must be called for each message read from the CAN bus.

Parameters
- **can_id** *(int)* – CAN-ID of the message
- **data** *(bytearray)* – Data part of the message (0 - 8 bytes)
- **timestamp** *(float)* – Timestamp of the message, preferably as a Unix timestamp

scanner = None
A NodeScanner for detecting nodes

1.2. API
send_message (can_id, data, remote=False)
Send a raw CAN message to the network.
This method may be overridden in a subclass if you need to integrate this library with a custom backend.
It is safe to call this from multiple threads.

Parameters
• can_id (int) – CAN-ID of the message
• data – Data to be transmitted (anything that can be converted to bytes)
• remote (bool) – Set to True to send remote frame

Raises can.CanError – When the message fails to be transmitted

send_periodic (can_id, data, period, remote=False)
Start sending a message periodically.

Parameters
• can_id (int) – CAN-ID of the message
• data – Data to be transmitted (anything that can be converted to bytes)
• period (float) – Seconds between each message
• remote (bool) – indicates if the message frame is a remote request to the slave node

Returns An task object with a .stop() method to stop the transmission

Return type canopen.network.PeriodicMessageTask

subscribe (can_id, callback)
Listen for messages with a specific CAN ID.

Parameters
• can_id (int) – The CAN ID to listen for.
• callback – Function to call when message is received.

unsubscribe (can_id, callback=None)
Stop listening for message.

Parameters
• can_id (int) – The CAN ID from which to unsubscribe.
• callback – If given, remove only this callback. Otherwise all callbacks for the CAN ID.

class canopen.RemoteNode (node_id, object_dictionary, load_od=False)
A CANopen remote node.

Parameters
• node_id (int) – Node ID (set to None or 0 if specified by object dictionary)
• object_dictionary (str, canopen.ObjectDictionary) – Object dictionary as either a path to a file, an ObjectDictionary or a file like object.
• load_od (bool) – Enable the Object Dictionary to be sent trough SDO’s to the remote node at startup.

id
The node id (1 - 127). Changing this after initializing the object will not have any effect.
**sdo**

The `canopen.sdo.SdoClient` associated with the node.

**sdo_channels**

List of available SDO channels (added with `add_sdo()`).

**tpdo**

The `canopen.pdo.PdoBase` for TPDO associated with the node.

**rpdo**

The `canopen.pdo.PdoBase` for RPDO associated with the node.

**nmt**

The `canopen.nmt.NmtMaster` associated with the node.

**emcy**

The `canopen.emcy.EmcyConsumer` associated with the node.

**object_dictionary**

The `canopen.ObjectDictionary` associated with the node

**network**

The `canopen.Network` owning the node

---

**add_sdo(rx_cobid, tx_cobid)**

Add an additional SDO channel.

The SDO client will be added to `sdo_channels`.

**Parameters**

- `rx_cobid (int)` – COB-ID that the server receives on
- `tx_cobid (int)` – COB-ID that the server responds with

**Returns**

The SDO client created

**Return type**

`canopen.sdo.SdoClient`

---

**curtis_hack = None**

Enable WORKAROUND for reversed PDO mapping entries

---

**load_configuration()**

Load the configuration of the node from the object dictionary.

**restore(subindex=1)**

Restore default parameters.

**Parameters**

`subindex (int)` – 1 = All parameters

2 = Communication related parameters

3 = Application related parameters

4 - 127 = Manufacturer specific

**store(subindex=1)**

Store parameters in non-volatile memory.

**Parameters**

`subindex (int)` – 1 = All parameters

2 = Communication related parameters

3 = Application related parameters

4 - 127 = Manufacturer specific

---

1.2. API
class canopen.LocalNode(node_id, object_dictionary)

    id
    The node id (1 - 127). Changing this after initializing the object will not have any effect.

    sdo
    The canopen.sdo.SdoServer associated with the node.

    object_dictionary
    The canopen.ObjectDictionary associated with the node

    network
    The canopen.Network owning the node

class canopen.network.MessageListener(network)

    Bases: object

    Lists for messages on CAN bus and feeds them to a Network instance.

    Parameters network (canopen.Network) – The network to notify on new messages.

class canopen.network.NodeScanner(network=None)

    Observes which nodes are present on the bus.

    Listens for the following messages:
        • Heartbeat (0x700)
        • SDO response (0x580)
        • TxPDO (0x180, 0x280, 0x380, 0x480)
        • EMCY (0x80)

    Parameters network (canopen.Network) – The network to use when doing active searching.

    active = True
    Activate or deactivate scanning

    nodes = None
    A list of nodes discovered

    reset()
    Clear list of found nodes.

    search(limit=127)
    Search for nodes by sending SDO requests to all node IDs.

class canopen.network.PeriodicMessageTask(can_id, data, period, bus, remote=False)

    Task object to transmit a message periodically using python-can’s CyclicSendTask

    Parameters
        • can_id (int) – CAN-ID of the message
        • data – Data to be transmitted (anything that can be converted to bytes)
        • period (float) – Seconds between each message
        • bus (can.BusABC) – python-can bus to use for transmission

    stop()
    Stop transmission
**update** *(data)*

Update data of message

**Parameters** `data` – New data to transmit
CANopen devices must have an object dictionary, which is used for configuration and communication with the device. An entry in the object dictionary is defined by:

- Index, the 16-bit address of the object in the dictionary
- Object type, such as an array, record, or simple variable
- Name, a string describing the entry
- Type, gives the datatype of the variable (or the datatype of all variables of an array)
- Attribute, which gives information on the access rights for this entry, this can be read/write (rw), read-only (ro) or write-only (wo)

The basic datatypes for object dictionary values such as booleans, integers and floats are defined in the standard, as well as composite datatypes such as strings, arrays and records. The composite datatypes can be subindexed with an 8-bit index; the value in subindex 0 of an array or record indicates the number of elements in the data structure, and is of type UNSIGNED8.

### 2.1 Supported formats

The currently supported file formats for specifying a node’s object dictionary are:

- EDS (standardized INI-file like format)
- DCF (same as EDS with bitrate and node ID specified)
- EPF (proprietary XML-format used by Inmotion Technologies)

### 2.2 Examples

The object dictionary file is normally provided when creating a node. Here is an example where the entire object dictionary gets printed out:
```python
node = network.add_node(6, 'od.eds')
for obj in node.object_dictionary.values():
    print('0x%X: %s' % (obj.index, obj.name))
    if isinstance(obj, canopen.objectdictionary.Record):
        for subobj in obj.values():
            print('%d: %s' % (subobj.subindex, subobj.name))
```

You can access the objects using either index/subindex or names:

```python
device_name_obj = node.object_dictionary['ManufacturerDeviceName']
vendor_id_obj = node.object_dictionary[0x1018][1]
```

## 2.3 API

### class canopen.ObjectDictionary

Representation of the object dictionary as a Python dictionary.

- `od[index]` : Return the object for the specified index (as int) or name (as string).
- `iter(od)` : Return an iterator over the indexes from the object dictionary.
- `index in od` : Return `True` if the index (as int) or name (as string) exists in the object dictionary.
- `len(od)` : Return the number of objects in the object dictionary.
- `values()` : Return a list of objects (records, arrays and variables).
- `add_object(obj)` : Add object to the object dictionary.

**Parameters** `obj` – Should be either one of `Variable, Record, or Array`.

- `bitrate = None` : Default bitrate if specified by file
- `get_variable(index, subindex=0)` : Get the variable object at specified index (and subindex if applicable).
  - **Returns** Variable if found, else `None`
  - **Return type** `canopen.objectdictionary.Variable`
- `node_id = None` : Node ID if specified by file

### class canopen.objectdictionary.Variable(name, index, subindex=0)

Simple variable.

- `len(var)` : Return the length of the variable data type in number of bits.
- `var == other` : Return `True` if the variables have the same index and subindex.
access_type = None
Access type, should be “rw”, “ro”, “wo”, or “const”

add_bit_definition(name, bits)
Associate bit(s) with a string description.

Parameters
• name (str) – Name of bit(s)
• bits (list) – List of bits as integers

add_value_description(value, descr)
Associate a value with a string description.

Parameters
• value (int) – Value to describe
• desc (str) – Description of value

bit_definitions = None
Dictionary of bitfield definitions

data_type = None
Data type according to the standard as an int

default = None
Default value at start-up

description = None
Description of variable

factor = None
Factor between physical unit and integer value

index = None
16-bit address of the object in the dictionary

max = None
Maximum allowed value

min = None
Minimum allowed value

name = None
String representation of the variable

parent = None
The ObjectDictionary, Record or Array owning the variable

subindex = None
8-bit sub-index of the object in the dictionary

unit = None
Physical unit

value = None
The value of this variable stored in the object dictionary

value_descriptions = None
Dictionary of value descriptions

class canopen.objectdictionary.Record(name, index)
Groups multiple Variable objects using subindices.
**record[subindex]**
   Return the *Variable* for the specified subindex (as int) or name (as string).

**iter(record)**
   Return an iterator over the subindexes from the record.

**subindex in record**
   Return True if the subindex (as int) or name (as string) exists in the record.

**len(record)**
   Return the number of subindexes in the record.

**record == other**
   Return True if the records have the same index.

**values()**
   Return a list of *Variable* in the record.

**add_member(variable)**
   Adds a *Variable* to the record.

**description = ''**
   Description for the whole record

**index = None**
   16-bit address of the record

**name = None**
   Name of record

**parent = None**
   The *ObjectDictionary* owning the record.

**class canopen.objectdictionary.Array(name, index)**
   An array of *Variable* objects using subindices.
   Actual length of array must be read from the node using SDO.

**array[subindex]**
   Return the *Variable* for the specified subindex (as int) or name (as string). This will work for all subindexes between 1 and 255. If the requested subindex has not been specified in the object dictionary, it will be created dynamically from the first subindex and suffixing the name with an underscore + the subindex in hex format.

**add_member(variable)**
   Adds a *Variable* to the record.

**description = ''**
   Description for the whole array

**index = None**
   16-bit address of the array

**name = None**
   Name of array

**parent = None**
   The *ObjectDictionary* owning the record.

**exception canopen.ObjectDictionaryError**
   Unsupported operation with the current Object Dictionary.
2.3.1 Constants

canopen.objectdictionary.UNSIGNED8

canopen.objectdictionary.UNSIGNED16

canopen.objectdictionary.UNSIGNED32

canopen.objectdictionary.UNSIGNED64

canopen.objectdictionary.INT8

canopen.objectdictionary.INT16

canopen.objectdictionary.INT32

canopen.objectdictionary.INT64

canopen.objectdictionary.BOOLEAN

canopen.objectdictionary.REAL32

canopen.objectdictionary.REAL64

canopen.objectdictionary.VISIBLE_STRING

canopen.objectdictionary.OCTET_STRING

canopen.objectdictionary.UNICODE_STRING

canopen.objectdictionary.DOMAIN

canopen.objectdictionary.SIGNED_TYPES

canopen.objectdictionary.UNSIGNED_TYPES

canopen.objectdictionary.INTEGER_TYPES

canopen.objectdictionary.FLOAT_TYPES

canopen.objectdictionary.NUMBER_TYPES

canopen.objectdictionary.DATA_TYPES
The NMT protocols are used to issue state machine change commands (e.g. to start and stop the devices), detect remote device bootups and error conditions.

The Module control protocol is used by the NMT master to change the state of the devices. The CAN-frame COB-ID of this protocol is always 0, meaning that it has a function code 0 and node ID 0, which means that every node in the network will process this message. The actual node ID, to which the command is meant to, is given in the data part of the message (at the second byte). This can also be 0, meaning that all the devices on the bus should go to the indicated state.

The Heartbeat protocol is used to monitor the nodes in the network and verify that they are alive. A heartbeat producer (usually a slave device) periodically sends a message with the binary function code of 1110 and its node ID (COB-ID = 0x700 + node ID). The data part of the frame contains a byte indicating the node status. The heartbeat consumer reads these messages.

CANopen devices are required to make the transition from the state Initializing to Pre-operational automatically during bootup. When this transition is made, a single heartbeat message is sent to the bus. This is the bootup protocol.

### 3.1 Examples

Access the NMT functionality using the `canopen.Node.nmt` attribute. Changing state can be done using the `state` attribute:

```python
node.nmt.state = 'OPERATIONAL'
# Same as sending NMT start
node.nmt.send_command(0x1)
```

You can also change state of all nodes simultaneously as a broadcast message:

```python
network.nmt.state = 'OPERATIONAL'
```

If the node transmits heartbeat messages, the `state` attribute gets automatically updated with current state:
# Send NMT start to all nodes
```python
network.send_message(0x0, [0x1, 0])
node.nmt.wait_for_heartbeat()
assert node.nmt.state == 'OPERATIONAL'
```

## 3.2 API

```python
class canopen.nmt.NmtMaster(node_id)
```

**add_heartbeat_callback** *(callback)*
Add function to be called on heartbeat reception.

**Parameters**

- **callback** – Function that should accept an NMT state as only argument.

**send_command** *(code)*
Send an NMT command code to the node.

**Parameters**

- **code** *(int)* – NMT command code.

**start_node_guarding** *(period)*
Starts the node guarding mechanism.

**Parameters**

- **period** *(float)* – Period (in seconds) at which the node guarding should be advertised to the slave node.

**stop_node_guarding** *
Stops the node guarding mechanism.

**timestamp** = None
Timestamp of last heartbeat message

**wait_for_bootup** *(timeout=10)*
Wait until a boot-up message is received.

**wait_for_heartbeat** *(timeout=10)*
Wait until a heartbeat message is received.

**exception** canopen.nmt.NmtError
Some NMT operation failed.
The SDO protocol is used for setting and for reading values from the object dictionary of a remote device. The device whose object dictionary is accessed is the SDO server and the device accessing the remote device is the SDO client. The communication is always initiated by the SDO client. In CANopen terminology, communication is viewed from the SDO server, so that a read from an object dictionary results in an SDO upload and a write to a dictionary entry is an SDO download.

Because the object dictionary values can be larger than the eight bytes limit of a CAN frame, the SDO protocol implements segmentation and desegmentation of longer messages. Actually, there are two of these protocols: SDO download/upload and SDO Block download/upload. The SDO block transfer is a newer addition to standard, which allows large amounts of data to be transferred with slightly less protocol overhead.

The COB-IDs of the respective SDO transfer messages from client to server and server to client can be set in the object dictionary. Up to 128 SDO servers can be set up in the object dictionary at addresses 0x1200 - 0x127F. Similarly, the SDO client connections of the device can be configured with variables at 0x1280 - 0x12FF. However the pre-defined connection set defines an SDO channel which can be used even just after bootup (in the Pre-operational state) to configure the device. The COB-IDs of this channel are 0x600 + node ID for receiving and 0x580 + node ID for transmitting.

4.1 Examples

SDO objects can be accessed using the .sdo member which works like a Python dictionary. Indexes and subindexes can be identified by either name or number. The code below only creates objects, no messages are sent or received yet:

```python
# Complex records
command_all = node.sdo['ApplicationCommands']['CommandAll']
actual_speed = node.sdo['ApplicationStatus']['ActualSpeed']
control_mode = node.sdo['ApplicationSetupParameters']['RequestedControlMode']

# Simple variables
device_type = node.sdo[0x1000]
```

(continues on next page)
# Arrays

canopen Documentation, Release 1.0.0

(continued from previous page)

to actually read or write the variables, use the .raw, .phys, .desc, or .bits attributes:

```python
print("The device type is 0x" + device_type.raw)

# Using value descriptions instead of integers (if supported by OD)
control_mode.desc = 'Speed Mode'

# Set individual bit
command_all.bits[3] = 1

# Read and write physical values scaled by a factor (if supported by OD)
print("The actual speed is " + actual_speed.phys)

# Iterate over arrays or records
for error in error_log.values():
    print("Error 0x" + error.raw)
```

it is also possible to read and write to variables that are not in the object dictionary, but only using raw bytes:

```python
device_type_data = node.sdo(upload(0x1000, 0)
node.sdo.download(0x1017, 0, b'\x00\x00')
```

variables can be opened as readable or writable file objects which can be useful when dealing with large amounts of data:

```python
# Open the Store EDS variable as a file like object
infile = node.sdo[0x1021].open('r', encoding='ascii')
# Open a file for writing to
outfile = open('out.eds', 'w', encoding='ascii')
# Iteratively read lines from node and write to file
outfile.writelines(infile)
# Clean-up
infile.close()
outfile.close()
```

most APIs accepting file objects should also be able to accept this.

Block transfer can be used to effectively transfer large amounts of data if the server supports it. This is done through the file object interface:

```python
FIRMWARE_PATH = '/path/to/firmware.bin'
FILESIZE = os.path.getsize(FIRMWARE_PATH)
infile = open(FIRMWARE_PATH, 'rb')
outfile = node.sdo['Firmware'].open('wb', size=FILESIZE, block_transfer=True)

while True:
    data = infile.read(1024)
    if not data:
        break
    outfile.write(data)
infile.close()
outfile.close()
```
Warning: Block transfer is still in experimental stage!

4.2 API

class canopen.sdo.SdoClient (rx_cobid, tx_cobid, od)
Handles communication with an SDO server.

Parameters

• rx_cobid (int) – COB-ID that the server receives on (usually 0x600 + node ID)
• tx_cobid (int) – COB-ID that the server responds with (usually 0x580 + node ID)
• od (canopen.ObjectDictionary) – Object Dictionary to use for communication

od
The canopen.ObjectDictionary associated with this object.

c[index]
Return the SDO object for the specified index (as int) or name (as string).

iter(c)
Return an iterator over the indexes from the object dictionary.

index in c
Return True if the index (as int) or name (as string) exists in the object dictionary.

len(c)
Return the number of indexes in the object dictionary.

values()
Return a list of objects (records, arrays and variables).

MAX_RETRIES = 1
Max number of request retries before raising error

PAUSE_BEFORE_SEND = 0.0
Seconds to wait before sending a request, for rate limiting

RESPONSE_TIMEOUT = 0.3
Max time in seconds to wait for response from server

abort (abort_code=134217728)
Abort current transfer.

download (index, subindex, data, force_segment=False)
May be called to make a write operation without an Object Dictionary.

Parameters

• index (int) – Index of object to write.
• subindex (int) – Sub-index of object to write.
• data (bytes) – Data to be written.
• force_segment (bool) – Force use of segmented transfer regardless of data size.

Raises

• canopen.SdoCommunicationError – On unexpected response or timeout.
open (index, subindex=0, mode='rb', encoding='ascii', buffering=1024, size=None, block_transfer=False, force_segment=False)
Open the data stream as a file like object.

Parameters

- **index (int)** – Index of object to open.
- **subindex (int)** – Sub-index of object to open.
- **mode (str)** – Character Meaning
  - 'r' open for reading (default)
  - 'w' open for writing
  - 'b' binary mode (default)
  - 't' text mode

- **encoding (str)** – The str name of the encoding used to decode or encode the file. This will only be used in text mode.
- **buffering (int)** – An optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer.
- **size (int)** – Size of data to that will be transmitted.
- **block_transfer (bool)** – If block transfer should be used.
- **force_segment (bool)** – Force use of segmented download regardless of data size.

Returns A file like object.

upload (index, subindex)
May be called to make a read operation without an Object Dictionary.

Parameters

- **index (int)** – Index of object to read.
- **subindex (int)** – Sub-index of object to read.

Returns A data object.

Return type bytes

Raises

- **canopen.SdoCommunicationError** – On unexpected response or timeout.

canopen.sdo.SdoServer (rx_cobid, tx_cobid, node)
Creates an SDO server.

Parameters

- **rx_cobid (int)** – COB-ID that the server receives on (usually 0x600 + node ID)
- **tx_cobid (int)** – COB-ID that the server responds with (usually 0x580 + node ID)
- **od (canopen.LocalNode)** – Node object owning the server

Chapter 4. Service Data Object (SDO)
od
   The canopen.ObjectDictionary associated with this object.

c[index]
   Return the SDO object for the specified index (as int) or name (as string).

iter(c)
   Return an iterator over the indexes from the object dictionary.

index in c
   Return True if the index (as int) or name (as string) exists in the object dictionary.

len(c)
   Return the number of indexes in the object dictionary.

values()
   Return a list of objects (records, arrays and variables).

abort(abort_code=134217728)
   Abort current transfer.

download(index, subindex, data, force_segment=False)
   May be called to make a write operation without an Object Dictionary.
   Parameters
   • index (int) – Index of object to write.
   • subindex (int) – Sub-index of object to write.
   • data (bytes) – Data to be written.

upload(index, subindex)
   May be called to make a read operation without an Object Dictionary.
   Parameters
   • index (int) – Index of object to read.
   • subindex (int) – Sub-index of object to read.
   Returns A data object.
   Return type bytes

class canopen.sdo.Variable(sdo_node, od)
   Access object dictionary variable values using SDO protocol.

od
   The canopen.objectdictionary.Variable associated with this object.

bits
   Access bits using integers, slices, or bit descriptions.

data
   Byte representation of the object as bytes.

desc
   Converts to and from a description of the value as a string.

open(mode='rb', encoding='ascii', buffering=1024, size=None, block_transfer=False)
   Open the data stream as a file like object.
Parameters

- **mode** *(str)* –
  
<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'r'</td>
<td>open for reading (default)</td>
</tr>
<tr>
<td>'w'</td>
<td>open for writing</td>
</tr>
<tr>
<td>'b'</td>
<td>binary mode (default)</td>
</tr>
<tr>
<td>'t'</td>
<td>text mode</td>
</tr>
</tbody>
</table>

- **encoding** *(str)* – The str name of the encoding used to decode or encode the file. This will only be used in text mode.

- **buffering** *(int)* – An optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer.

- **size** *(int)* – Size of data to that will be transmitted.

- **block_transfer** *(bool)* – If block transfer should be used.

Returns  A file like object.

**phys**

Physical value scaled with some factor (defaults to 1).

On object dictionaries that support specifying a factor, this can be either a *float* or an *int*. Non integers will be passed as is.

**raw**

Raw representation of the object.

This table lists the translations between object dictionary data types and Python native data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>bool</td>
</tr>
<tr>
<td>UNSIGNEDxx</td>
<td>int</td>
</tr>
<tr>
<td>INTEGERxx</td>
<td>int</td>
</tr>
<tr>
<td>REALxx</td>
<td>float</td>
</tr>
<tr>
<td>VISIBLE_STRING</td>
<td>str/unicode (Python 2)</td>
</tr>
<tr>
<td>UNICODE_STRING</td>
<td>str/unicode (Python 2)</td>
</tr>
<tr>
<td>OCTET_STRING</td>
<td>bytes</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>bytes</td>
</tr>
</tbody>
</table>

Data types that this library does not handle yet must be read and written as *bytes*.

**read** *(fmt='raw')*

Alternative way of reading using a function instead of attributes.

May be useful for asynchronous reading.

Parameters  **fmt** *(str)* –

How to return the value

- 'raw'
- 'phys'
- 'desc'
Returns The value of the variable.

```python
write(value, fmt='raw')
```
Alternative way of writing using a function instead of attributes.
May be useful for asynchronous writing.

**Parameters**

- **fmt (str)**: How to write the value
  - 'raw'
  - 'phys'
  - 'desc'

```python
class canopen.sdo.Record(sdo_node, od)
```

**od**
The `canopen.objectdictionary.Record` associated with this object.

```python
record[subindex]
```
Return the `canopen.sdo.Variable` for the specified subindex (as int) or name (as string).

```python
iter(record)
```
Return an iterator over the subindexes from the record.

```python
subindex in record
```
Return `True` if the subindex (as int) or name (as string) exists in the record.

```python
len(record)
```
Return the number of subindexes in the record.

```python
values()
```
Return a list of `canopen.sdo.Variable` in the record.

```python
class canopen.sdo.Array(sdo_node, od)
```

**od**
The `canopen.objectdictionary.Array` associated with this object.

```python
array[subindex]
```
Return the `canopen.sdo.Variable` for the specified subindex (as int) or name (as string).

```python
iter(array)
```
Return an iterator over the subindexes from the array. This will make a SDO read operation on subindex 0 in order to get the actual length of the array.

```python
subindex in array
```
Return `True` if the subindex (as int) or name (as string) exists in the array. This will make a SDO read operation on subindex 0 in order to get the actual length of the array.

```python
len(array)
```
Return the length of the array. This will make a SDO read operation on subindex 0.

```python
values()
```
Return a list of `canopen.sdo.Variable` in the array. This will make a SDO read operation on subindex 0 in order to get the actual length of the array.

```python
exception canopen.SdoAbortedError(code)
```
Bases: `canopen.sdo.exceptions.SdoError`
SDO abort exception.

```
code = None
Abort code
```

**exception** `canopen.SdoCommunicationError`

**Bases:** `canopen.sdo.exceptions.SdoError`

No or unexpected response from slave.
The Process Data Object protocol is used to process real time data among various nodes. You can transfer up to 8 bytes (64 bits) of data per one PDO either from or to the device. One PDO can contain multiple object dictionary entries and the objects within one PDO are configurable using the mapping and parameter object dictionary entries.

There are two kinds of PDOs: transmit and receive PDOs (TPDO and RPDO). The former is for data coming from the device and the latter is for data going to the device; that is, with RPDO you can send data to the device and with TPDO you can read data from the device. In the pre-defined connection set there are identifiers for four (4) TPDOs and four (4) RPDOs available. With configuration 512 PDOs are possible.

PDOs can be sent synchronously or asynchronously. Synchronous PDOs are sent after the SYNC message whereas asynchronous messages are sent after internal or external trigger. For example, you can make a request to a device to transmit TPDO that contains data you need by sending an empty TPDO with the RTR flag (if the device is configured to accept TPDO requests).

With RPDOs you can, for example, start two devices simultaneously. You only need to map the same RPDO into two or more different devices and make sure those RPDOs are mapped with the same COB-ID.

### 5.1 Examples

A `canopen.RemoteNode` has `canopen.RemoteNode.rpdo` and `canopen.RemoteNode.tpdo` attributes that can be used to interact with the node using PDOs. These can be subindexed to specify which map to use (first map starts at 1, not 0):

```python
# Read current PDO configuration
node.tpdo.read()
node.rpdo.read()

# Do some changes to TPDO4 and RPDO4
node.tpdo[4].clear()
node.tpdo[4].add_variable('Application Status', 'Status All')
node.tpdo[4].add_variable('Application Status', 'Actual Speed')
node.tpdo[4].trans_type = 254
```

(continues on next page)
node.tpdo[4].event_timer = 10
node.tpdo[4].enabled = True

node.rpdo[4].clear()
node.rpdo[4].add_variable('Application Commands', 'Command All')
node.rpdo[4].add_variable('Application Commands', 'Command Speed')
node.rpdo[4].enabled = True

# Save new configuration (node must be in pre-operational)
node.nmt.state = 'PRE-OPERATIONAL'
nodel.tpdo.save()
nodel.rpdo.save()

# Start RPDO4 with an interval of 100 ms
node.rpdo[4]['Application Commands.Command Speed'].phys = 1000
node.rpdo[4].start(0.1)
nodel.nmt.state = 'OPERATIONAL'

# Read 50 values of speed and save to a file
with open('output.txt', 'w') as f:
    for i in range(50):
        node.tpdo[4].wait_for_reception()
        speed = node.tpdo['Application Status.Actual Speed'].phys
        f.write(' %s
' % speed)

# Using a callback to asynchronously receive values
# Do not do any blocking operations here!
def print_speed(message):
    print(' %s received' % message.name)
    for var in message:
        print(' %s = %d' % (var.name, var.raw))

node.tpdo[4].add_callback(print_speed)
time.sleep(5)

# Stop transmission of RPDO
node.rpdo[4].stop()

## 5.2 API

class canopenpdo.PdoBase(node)
    Represents the base implementation for the PDO object.

    Parameters node(object) – Parent object associated with this PDO instance

    pdo[no]
    Return the canopenpdo.Map for the specified map number. First map starts at 1.

    iter(pdo)
    Return an iterator of the available map numbers.

    len(pdo)
    Return the number of supported maps.

    export (filename)
    Export current configuration to a database file.
Parameters `filename (str)` – Filename to save to (e.g. DBC, DBF, ARXML, KCD etc)

Returns The CanMatrix object created

Return type `canmatrix.canmatrix.CanMatrix`

`read()`
Read PDO configuration from node using SDO.

`save()`
Save PDO configuration to node using SDO.

`stop()`
Stop all running tasks.

**class canopenpdo.Map(pdo_node, com_record, map_array)**
One message which can have up to 8 bytes of variables mapped.

`map[name]`
Return the `canopenpdo.Variable` for the variable specified as "Group.Variable" or "Variable" or as a position starting at 0.

`iter(map)`
Return an iterator of the `canopenpdo.Variable` entries in the map.

`len(map)`
Return the number of variables in the map.

`add_callback(callback)`
Add a callback which will be called on receive.

Parameters `callback` – The function to call which must take one argument of a `Map`.

`add_variable(index, subindex=0, length=None)`
Add a variable from object dictionary as the next entry.

Parameters

- `index (str or int)` – Index of variable as name or number
- `subindex (str or int)` – Sub-index of variable as name or number
- `length (int)` – Size of data in number of bits

Returns Variable that was added

Return type `canopenpdo.Variable`

`clear()`
Clear all variables from this map.

`cob_id = None`
COB-ID for this PDO

`data = None`
Current message data

`enabled = None`
If this map is valid

`event_timer = None`
Event timer (optional) (in ms)

`inhibit_time = None`
Inhibit Time (optional) (in 100us)
map = None
List of variables mapped to this PDO

ame
A descriptive name of the PDO.

Examples:
- TxPDO1_node4
- RxPDO4_node1
- Unknown

period = None
Period of receive message transmission in seconds

predefined_cob_id = None
Default COB-ID if this PDO is part of the pre-defined connection set

read()
Read PDO configuration for this map using SDO.

remote_request()
Send a remote request for the transmit PDO. Silently ignore if not allowed.

rtr_allowed = None
Is the remote transmit request (RTR) allowed for this PDO

save()
Save PDO configuration for this map using SDO.

start(period=None)
Start periodic transmission of message in a background thread.

    Parameters period (float) – Transmission period in seconds

stop()
Stop transmission.

sync_start_value = None
Ignores SYNC objects up to this SYNC counter value (optional)

timestamp = None
Timestamp of last received message

trans_type = None
Transmission type (0-255)

transmit()
Transmit the message once.

update()
Update periodic message with new data.

wait_for_reception(timeout=10)
Wait for the next transmit PDO.

    Parameters timeout (float) – Max time to wait in seconds.
    Returns  Timestamp of message received or None if timeout.
    Return type  float

class canopen.pdo.Variable(od)
One object dictionary variable mapped to a PDO.
The `canopen.objectdictionary.Variable` associated with this object.

**bits**
Access bits using integers, slices, or bit descriptions.

**data**
Byte representation of the object as `bytes`.

**desc**
Converts to and from a description of the value as a string.

**get_data**
Reads the PDO variable from the last received message.

Returns: Variable value as `bytes`.

Return type: `bytes`

**offset = None**
Location of variable in the message in bits

**pdo_parent = None**
PDO object that is associated with this Variable Object

**phys**
Physical value scaled with some factor (defaults to 1).

On object dictionaries that support specifying a factor, this can be either a `float` or an `int`. Non integers will be passed as is.

**raw**
Raw representation of the object.

This table lists the translations between object dictionary data types and Python native data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td><code>bool</code></td>
</tr>
<tr>
<td>UNSIGNEDxx</td>
<td><code>int</code></td>
</tr>
<tr>
<td>INTEGERxx</td>
<td><code>int</code></td>
</tr>
<tr>
<td>REALxx</td>
<td><code>float</code></td>
</tr>
<tr>
<td>VISIBLE_STRING</td>
<td><code>str/unicode (Python 2)</code></td>
</tr>
<tr>
<td>UNICODE_STRING</td>
<td><code>str/unicode (Python 2)</code></td>
</tr>
<tr>
<td>OCTET_STRING</td>
<td><code>bytes</code></td>
</tr>
<tr>
<td>DOMAIN</td>
<td><code>bytes</code></td>
</tr>
</tbody>
</table>

Data types that this library does not handle yet must be read and written as `bytes`.

**read(fmt='raw')**
Alternative way of reading using a function instead of attributes.

May be useful for asynchronous reading.

Parameters: `fmt (str)` —

How to return the value

- ‘raw’
- ‘phys’
- ‘desc’
Returns The value of the variable.

set_data(data)
Set for the given variable the PDO data.

Parameters data(bytes) – Value for the PDO variable in the PDO message as bytes.

write(value, fmt='raw')
Alternative way of writing using a function instead of attributes.
May be useful for asynchronous writing.

Parameters fmt(str) –

How to write the value
• 'raw'
• 'phys'
• 'desc'
Synchronization Object (SYNC)

The Sync-Producer provides the synchronization-signal for the Sync-Consumer. When the Sync-Consumer receive the signal they start carrying out their synchronous tasks.

In general, the fixing of the transmission time of synchronous PDO messages coupled with the periodicity of transmission of the Sync Object guarantees that sensor devices may arrange to sample process variables and that actuator devices may apply their actuation in a coordinated fashion.

The identifier of the Sync Object is available at index 1005h.

6.1 Examples

Use the `canopen.Network.sync` attribute to start and stop the SYNC message:

```python
# Transmit every 10 ms
network.sync.start(0.01)

network.sync.stop()
```

6.2 API

class `canopen.sync.SyncProducer` *(network)*

Transmits a SYNC message periodically.

- **cob_id** = 128
  
  COB-ID of the SYNC message

- **start**(period=None)
  
  Start periodic transmission of SYNC message in a background thread.

  Parameters **period** *(float)* – Period of SYNC message in seconds.
stop()
   Stop periodic transmission of SYNC message.

transmit (count=None)
   Send out a SYNC message once.

   Parameters count (int) – Counter to add in message.
Emergency messages are triggered by the occurrence of a device internal fatal error situation and are transmitted from the concerned application device to the other devices with high priority. This makes them suitable for interrupt type error alerts. An Emergency Telegram may be sent only once per 'error event', i.e. the emergency messages must not be repeated. As long as no new errors occur on a device no further emergency message must be sent. By means of CANopen Communication Profile defined emergency error codes, the error register and device specific additional information are specified in the device profiles.

### 7.1 Examples

To list the currently active emergencies for a particular node, one can use the `.active` attribute which is a list of `canopen.emcy.EmcyError` objects:

```python
active_codes = [emcy.code for emcy in node.emcy.active]
all_codes = [emcy.code for emcy in node.emcy.log]
```

The `canopen.emcy.EmcyError` objects are actually exceptions so that they can be easily raised if that’s what you want:

```python
if node.emcy.active:
    raise node.emcy.active[-1]
```

### 7.2 API

```python
class canopen.emcy.EmcyConsumer

    active = None
    Only active EMCYs. Will be cleared on Error Reset
```
add_callback(callback)
    Get notified on EMCY messages from this node.

    Parameters callback – Callable which must take one argument of an EmcyError instance.

log = None
    Log of all received EMCYs for this node

reset()
    Reset log and active lists.

wait(emcy_code=None, timeout=10)
    Wait for a new EMCY to arrive.

    Parameters
    • emcy_code (int) – EMCY code to wait for
    • timeout (float) – Max time in seconds to wait

    Returns The EMCY exception object or None if timeout

    Return type canopen.emcy.EmcyError

exception canopen.emcy.EmcyError (code, register, data, timestamp)
    EMCY exception.

    code = None
        EMCY code

    data = None
        Vendor specific data

    register = None
        Error register

    timestamp = None
        Timestamp of message
Time Stamp Object (TIME)

Usually the Time-Stamp object represents an absolute time in milliseconds after midnight and the number of days since January 1, 1984. This is a bit sequence of length 48 (6 bytes).

8.1 API

```python
class canopen.timestamp.TimeProducer(network)
    Produces timestamp objects.
    
    cob_id = 256
    COB-ID of the SYNC message

    transmit (timestamp=None)
    Send out the TIME message once.

    Parameters timestamp (float) – Optional Unix timestamp to use, otherwise the current time is used.
```
The LSS protocol is used to change the node id and baud rate of the target CANOpen device (slave). To change these values, configuration state should be set first by master. Then modify the node id and the baud rate. There are two options to switch from waiting state to configuration state. One is to switch all the slave at once, the other way is to switch only one slave. The former can be used to set baud rate for all the slaves. The latter can be used to change node id one by one.

Once you finished the setting, the values should be saved to non-volatile memory. Finally, you can switch to LSS waiting state.

**Note:** Some method and constance names are changed:

```python
send_switch_mode_global() ==> send_switch_state_global()
network.lss.CONFIGURATION_MODE ==> network.lss.CONFIGURATION_STATE
network.lss.NORMAL_MODE ==> network.lss.WAITING_STATE
```

You can still use the old name, but please use the new names.

**Note:** Fastscan is supported from v0.8.0. LSS identify slave service is not implemented.

### 9.1 Examples

Switch all the slave into CONFIGURATION state. There is no response for the message.

```python
network.lss.send_switch_state_global(network.lss.CONFIGURATION_STATE)
```

Or, you can call this method with 4 IDs if you want to switch only one slave:
vendorId = 0x00000022
productCode = 0x12345678
revisionNumber = 0x0000555
serialNumber = 0x00abcdef

ret_bool = network.lss.send_switch_state_selective(vendorId, productCode,
revisionNumber, serialNumber)

Or, you can run fastscan procedure

ret_bool, lss_id_list = network.lss.fast_scan()

Once one of sensors goes to CONFIGURATION state, you can read the current node id of the LSS slave:

node_id = network.lss.inquire_node_id()

Change the node id and baud rate:

network.lss.configure_node_id(node_id+1)
network.lss.configure_bit_timing(2)

This is the table for converting the argument index of bit timing into baud rate.

<table>
<thead>
<tr>
<th>idx</th>
<th>Baud rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 MBit/sec</td>
</tr>
<tr>
<td>1</td>
<td>800 kBit/sec</td>
</tr>
<tr>
<td>2</td>
<td>500 kBit/sec</td>
</tr>
<tr>
<td>3</td>
<td>250 kBit/sec</td>
</tr>
<tr>
<td>4</td>
<td>125 kBit/sec</td>
</tr>
<tr>
<td>5</td>
<td>100 kBit/sec</td>
</tr>
<tr>
<td>6</td>
<td>50 kBit/sec</td>
</tr>
<tr>
<td>7</td>
<td>20 kBit/sec</td>
</tr>
<tr>
<td>8</td>
<td>10 kBit/sec</td>
</tr>
</tbody>
</table>

Save the configuration:

network.lss.store_configuration()

Finally, you can switch the state of the slave(s) from CONFIGURATION state to WAITING state:

network.lss.send_switch_state_global(network.lss.WAITING_STATE)

9.2 API

class canopen.lss.LssMaster
The Master of Layer Setting Services

    RESPONSE_TIMEOUT = 0.5
        Max time in seconds to wait for response from server

    activate_bit_timing(switch_delay_ms)
        Activate the bit timing.
**Parameters**

**switch_delay_ms**(uint16_t) – The slave that receives this message waits for switch delay, then activate the bit timing. But it shouldn’t send any message until another switch delay is elapsed.

**configure_bit_timing**(new_bit_timing)
Set the bit timing.

**Parameters**

**new_bit_timing**(int) – bit timing index. 0: 1 MBit/sec, 1: 800 kBit/sec, 2: 500 kBit/sec, 3: 250 kBit/sec, 4: 125 kBit/sec 5: 100 kBit/sec, 6: 50 kBit/sec, 7: 20 kBit/sec, 8: 10 kBit/sec

**configure_node_id**(new_node_id)
Set the node id

**Parameters**

**new_node_id**(int) – new node id to set

**fast_scan**()
This command sends a series of fastscan message to find unconfigured slave with lowest number of LSS identities

**Returns** True if a slave is found. False if there is no candidate. list is the LSS identities [vendor_id, product_code, revision_number, seerial_number]

**Return type** bool, list

**inquire_lss_address**(req_cs)
Read the part of LSS address. VENDOR_ID, PRODUCT_CODE, REVISION_NUMBER, or SERIAL_NUMBER

**Parameters**

**req_cs**(int) – command specifier for request

**Returns** part of LSS address

**Return type** int

**inquire_node_id**()
Read the node id. CANopen node id must be within the range from 1 to 127.

**Returns** node id. 0 means it is not read by LSS protocol

**Return type** int

**send_identify_remote_slave**(vendorId, productCode, revisionNumberLow, revisionNumberHigh, serialNumberLow, serialNumberHigh)
This command sends the range of LSS address to find the slave nodes in the specified range

**Parameters**

- **vendorId**(int)
- **productCode**(int)
- **revisionNumberLow**(int)
- **revisionNumberHigh**(int)
- **serialNumberLow**(int)
- **serialNumberHigh**(int)

**Returns** True if any slave responds. False if there is no response.

**Return type** bool
**send_switch_mode_global** *(mode)*
obsolete

**send_switch_state_global** *(mode)*
switch mode to CONFIGURATION_STATE or WAITING_STATE in the all slaves on CAN bus. There is no reply for this request

Parameters

- `mode (int)` – CONFIGURATION_STATE or WAITING_STATE

**send_switch_state_selective** *(vendorId, productCode, revisionNumber, serialNumber)*
switch mode from WAITING_STATE to CONFIGURATION_STATE only if 128bits LSS address matches with the arguments. It sends 4 messages for each argument. Then wait the response from the slave. There will be no response if there is no matching slave

Parameters

- `vendorId (int)` – object index 0x1018 subindex 1
- `productCode (int)` – object index 0x1018 subindex 2
- `revisionNumber (int)` – object index 0x1018 subindex 3
- `serialNumber (int)` – object index 0x1018 subindex 4

Returns True if any slave responds. False if there is no response.

Return type `bool`

**store_configuration** ()
Store node id and baud rate.

**class canopen.lss.LssError**

Bases: `exceptions.Exception`

Some LSS operation failed.
Integration with existing code

Sometimes you need to use this library together with some existing code base or you have CAN drivers not supported by python-can. This chapter will cover some use cases.

10.1 Re-using a bus

If you need to interact with the CAN-bus outside of this library too and you want to use the same python-can Bus instance, you need to tell the Network which Bus to use and also add the `canopen.network.MessageListener` to your existing `can.Notifier`.

Here is a short example:

```python
import canopen
import can

# A Bus instance created outside
bus = can.interface.Bus()

network = canopen.Network()
# Associate the bus with the network
network.bus = bus

# Add your list of can.Listener with the network's
listeners = [can.Printer()] + network.listeners
# Start the notifier
notifier = can.Notifier(bus, listeners, 0.5)
```

10.2 Using a custom backend

If the python-can package does not have support for your CAN interface then you need to create a sub-class of `canopen.Network` and provide your own means of sending messages. You also need to feed incoming messages...
in a background thread to `canopen.Network.notify()`.

Here is an example:

```python
import canopen

class CustomNetwork(canopen.Network):
    def connect(self, *args, **kwargs):
        # Optionally use this to start communication with CAN
        pass

    def disconnect(self):
        # Optionally use this to stop communication
        pass

    def send_message(self, can_id, data, remote=False):
        # Send the message with the 11-bit can_id and data which might be
        # a bytearray or list of integers.
        # if remote is True then it should be sent as an RTR.
        pass

network = CustomNetwork()

# Should be done in a thread but here we notify the network for
# demonstration purposes only
network.notify(0x701, bytearray([0x05]), time.time())
```
On top of the standard CANopen functionality which includes the DS301 application layer there can be additional profiles specifically for certain applications.

## 11.1 CiA 402 CANopen device profile for motion controllers and drives

This device profile has a control state machine for controlling the behaviour of the drive. Therefore one needs to instantiate a node with the `Node402` class.

Create a node with `Node402`:

```python
import canopen
from canopen.profiles.p402 import Node402

some_node = canopen.Node402(3, 'someprofile.eds')
network = canopen.Network()
network.add_node(some_node)
```

### 11.1.1 The Power State Machine

The `PowerStateMachine` class provides the means of controlling the states of this state machine. The static method `on_PDO1_callback()` is added to the TPDO1 callback.

State changes can be controlled by writing a specific value to register 0x6040, which is called the “Controlword”. The current status can be read from the device by reading the register 0x6041, which is called the “Statusword”. Changes in state can only be done in the ‘OPERATIONAL’ state of the NmtMaster.

TPDO1 needs to be set up correctly. For this, run the the `Node402.setup_402_state_machine()` method. Note that this setup routine will change only TPDO1 and automatically go to the ‘OPERATIONAL’ state of the NmtMaster.
# run the setup routine for TPDO1 and it’s callback
some_node.setup_402_state_machine()

Write Controlword and read Statusword:

```python
# command to go to 'READY TO SWITCH ON' from 'NOT READY TO SWITCH ON' or 'SWITCHED ON'
some_node.sdo[0x6040].raw = 0x06

# Read the state of the Statusword
some_node.sdo[0x6041].raw
```

During operation the state can change to states which cannot be commanded by the Controlword, for example a ‘FAULT’ state. Therefore the **PowerStateMachine** class (in similarity to the **NmtMaster** class) automatically monitors state changes of the Statusword which is sent by TPDO1. The available callback on the TPDO1 will then extract the information and mirror the state change in the **Node402.powerstate_402** attribute.

Similar to the **NmtMaster** class, the states of the **Node402** class _state_ attribute can be read and set (command) by a string:

```python
# command a state (an SDO message will be called)
some_node.powerstate_402.state = 'SWITCHED ON'
# read the current state
some_node.powerstate_402.state = 'SWITCHED ON'
```

Available states:
- ‘NOT READY TO SWITCH ON’
- ‘SWITCH ON DISABLED’
- ‘READY TO SWITCH ON’
- ‘SWITCHED ON’
- ‘OPERATION ENABLED’
- ‘FAULT’
- ‘FAULT REACTION ACTIVE’
- ‘QUICK STOP ACTIVE’

Available commands
- ‘SWITCH ON DISABLED’
- ‘DISABLE VOLTAGE’
- ‘READY TO SWITCH ON’
- ‘SWITCHED ON’
- ‘OPERATION ENABLED’
- ‘QUICK STOP ACTIVE’
A
abort () (canopen.sdo.SdoClient method), 21
abort () (canopen.sdo.SdoServer method), 23
access_type (canopen.objectdictionary.Variable attribute), 12
activate_bit_timing () (canopen.lss.LssMaster method), 40
active (canopen.emcy.EmcyConsumer attribute), 35
active (canopen.network.NodeScanner attribute), 8
add_bit_definition ()
  (canopen.objectdictionary.Variable method), 13
add_callback () (canopen.emcy.EmcyConsumer method), 35
add_callback () (canopen.pdo.Map method), 29
add_hearbeat_callback ()
  (canopen.nmt.NmtMaster method), 18
add_member ()
  (canopen.objectdictionary.Array method), 14
add_member ()
  (canopen.objectdictionary.Record method), 14
add_node () (canopen.Network method), 4
add_object () (canopen.ObjectDictionary method), 12
add_sdo () (canopen.RemoteNode method), 7
add_value_description ()
  (canopen.objectdictionary.Variable method), 13
add_variable () (canopen.pdo.Map method), 29
Array (class in canopen.objectdictionary), 14
Array (class in canopen.sdo), 25

B
bit_definitions (canopen.objectdictionary.Variable attribute), 13
bitrate (canopen.ObjectDictionary attribute), 12
bits (canopen.pdo.Variable attribute), 31
bits (canopen.sdo.Variable attribute), 23
bus (canopen.Network attribute), 5

canopen.objectdictionary.BOOLEAN (built-in variable), 15
canopen.objectdictionary.DATA_TYPES (built-in variable), 15
canopen.objectdictionary.DOMAIN (built-in variable), 15
canopen.objectdictionary.FLOAT_TYPES (built-in variable), 15
canopen.objectdictionary.INTEGER16 (built-in variable), 15
canopen.objectdictionary.INTEGER32 (built-in variable), 15
canopen.objectdictionary.INTEGER64 (built-in variable), 15
canopen.objectdictionary.INTEGER_TYPES (built-in variable), 15
canopen.objectdictionary.NUMBER_TYPES (built-in variable), 15
canopen.objectdictionary.OCTET_STRING (built-in variable), 15
canopen.objectdictionary.REAL32 (built-in variable), 15
canopen.objectdictionary.REAL64 (built-in variable), 15
canopen.objectdictionary.SIGNED_TYPES (built-in variable), 15
canopen.objectdictionary.UNICODE_STRING (built-in variable), 15
canopen.objectdictionary.UNSIGNED16 (built-in variable), 15
canopen.objectdictionary.UNSIGNED32 (built-in variable), 15
canopen.objectdictionary.UNSIGNED64 (built-in variable), 15
canopen.objectdictionary.UNSIGNED8 (built-in variable), 15
canopen.objectdictionary.UNSIGNED_TYPES  
(built-in variable), 15

canopen.objectdictionary.VISIBLE_STRING  
(built-in variable), 15

check() (canopen.Network method), 5

clear() (canopen.pdo.Map method), 29

cob_id (canopen.pdo.Map attribute), 29

cob_id (canopen.sync.SyncProducer attribute), 33

cob_id (canopen.timestamp.TimeProducer attribute), 37

configure_bit_timing() (canopen.lss.LssMaster method), 41

configure_node_id() (canopen.lss.LssMaster method), 41

connect() (canopen.Network method), 5

create_node() (canopen.Network method), 5

curtis_hack (canopen.RemoteNode attribute), 7

D

data (canopen.emcy.EmcyError attribute), 36

data (canopen.pdo.Map attribute), 29

data (canopen.pdo.Variable attribute), 31

data (canopen.sdo.Variable attribute), 23

data_type (canopen.objectdictionary.Variable attribute), 13

default (canopen.objectdictionary.Variable attribute), 13

desc (canopen.pdo.Variable attribute), 31

desc (canopen.sdo.Variable attribute), 23

description (canopen.objectdictionary.Array attribute), 14

description (canopen.objectdictionary.Record attribute), 14

description (canopen.objectdictionary.Variable attribute), 13

disconnect() (canopen.Network method), 5

download() (canopen.sdo.SdoClient method), 21

download() (canopen.sdo.SdoServer method), 23

E

ecmy (canopen.RemoteNode attribute), 7

EmcyConsumer (class in canopen.emcy), 35

EmcyError, 36

enabled (canopen.pdo.Map attribute), 29

event_timer (canopen.pdo.Map attribute), 29

export () (canopen.pdo.PdoBase method), 28

F

factor (canopen.objectdictionary.Variable attribute), 13

fast_scan() (canopen.lss.LssMaster method), 41

get_data() (canopen.pdo.Variable method), 31

get_variable() (canopen.ObjectDictionary method), 12

I

id (canopen.LocalNode attribute), 8

id (canopen.RemoteNode attribute), 6

index (canopen.objectdictionary.Array attribute), 14

index (canopen.objectdictionary.Record attribute), 14

index (canopen.objectdictionary.Variable attribute), 13

inhibit_time (canopen.pdo.Map attribute), 29

inquire_lss_address() (canopen.lss.LssMaster method), 41

inquire_node_id() (canopen.lss.LssMaster method), 41

listeners (canopen.Network attribute), 5

load_configuration() (canopen.RemoteNode method), 7

LocalNode (class in canopen), 7

log (canopen.emcy.EmcyConsumer attribute), 36

LssError (class in canopen.lss), 42

LssMaster (class in canopen.lss), 40

M

map (canopen.pdo.Map attribute), 29

Map (class in canopen.pdo), 29

max (canopen.objectdictionary.Variable attribute), 13

MAX_RETRIES (canopen.sdo.SdoClient attribute), 21

MessageListener (class in canopen.network), 8

min (canopen.objectdictionary.Variable attribute), 13

N

name (canopen.objectdictionary.Array attribute), 14

name (canopen.objectdictionary.Record attribute), 14

name (canopen.objectdictionary.Variable attribute), 13

name (canopen.pdo.Map attribute), 30

network (canopen.LocalNode attribute), 8

network (canopen.RemoteNode attribute), 7

Network (class in canopen), 4

nmt (canopen.Network attribute), 4

nmt (canopen.RemoteNode attribute), 7

NmtError, 18

NmtMaster (class in canopen.nmt), 18

node_id (canopen.ObjectDictionary attribute), 12

nodes (canopen.network.NodeScanner attribute), 8

NodeScanner (class in canopen.network), 8

notify() (canopen.Network method), 5

O

object_dictionary (canopen.LocalNode attribute), 8
transmit() (canopenpdo.Map method), 30
transmit() (canopen.sync.SyncProducer method), 34
transmit() (canopen.timestamp.TimeProducer method), 37

U
unit (canopen.objectdictionary.Variable attribute), 13
unsubscribe() (canopen.Network method), 6
update() (canopen.network.PeriodicMessageTask method), 8
update() (canopenpdo.Map method), 30
upload() (canopen.sdo.SdoClient method), 22
upload() (canopen.sdo.SdoServer method), 23

V
value (canopen.objectdictionary.Variable attribute), 13
value_descriptions (canopen.objectdictionary.Variable attribute), 13
values() (canopen.Network method), 4
values() (canopen.ObjectDictionary method), 12
values() (canopen.objectdictionary.Record method), 14
values() (canopen.sdo.Array method), 25
values() (canopen.sdo.Record method), 25
values() (canopen.sdo.SdoClient method), 21
values() (canopen.sdo.SdoServer method), 23
Variable (class in canopen.objectdictionary), 12
Variable (class in canopen pdo), 30
Variable (class in canopen.sdo), 23

W
wait() (canopen.emcy.EmcyConsumer method), 36
wait_for_boot_up() (canopen.nmt.NmtMaster method), 18
wait_for_heartbeat() (canopen.nmt.NmtMaster method), 18
wait_for_reception() (canopenpdo.Map method), 30
write() (canopenpdo.Variable method), 32
write() (canopen.sdo.Variable method), 25