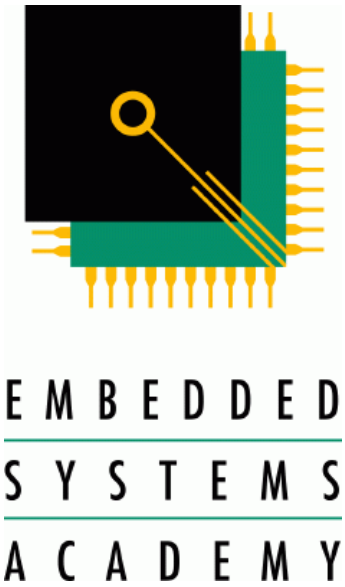


ESAcademy Recommended Practice CANopen API

W O R K D R A F T

Rev 0.6 of August 18th, 2003 – First Published Version

Table of Content

Table of Content 2
1 Motivation and Introduction 3
1.1 Definition of Terms 3
1.2 Block Diagram 5
2 Interface Implementation Options 6
2.1 Interfacing with Programming Languages 6
2.2 Interfacing with a Message System 6
3 Using a Process Image 7
3.1 The CANopen API Process Image 7
3.2 Configuring the Usage of the Process Image 7
3.3 Accessing the Process Image 7
4 CANopen API Functions 8
4.1 Requests from Host to CANopen Task 8
4.1.1 Function GetID 8
4.1.2 Function GetStatus 9
4.1.3 Function InitCANopen 10
4.1.4 Function ReadProcessData 11
4.1.5 Function WriteProcessData 12
4.2 Indications from CANopen Task to Host 13
4.2.1 Callback Function NMTChange 13
4.2.2 Callback Function ReceivedData 14
4.2.3 Callback Function FatalError 15
5 CANopen Task Setup File 16
5.1 Generic Setup File Format 16
5.2 Entries in the Setup File 16
5.2.1 Entry [ID] 17
5.2.2 Entry [NODE] 17
5.2.3 Entry [RPDO] 18
5.2.4 Entry [TPDO] 19
5.2.5 Entry [COD] 20
5.2.6 Entry [RWOD] 21
5.2.7 Entry [PIMG] 23

1 Motivation and Introduction

The networking standard CANopen does not specify or standardize an application programming interface (API) for accessing the CANopen network protocol stack of CANopen nodes. Most existing CANopen implementations use their own API and as a result once a specific CANopen solution is implemented, there is no easy migration path to port an existing CANopen application to a different CANopen protocol stack.

This document describes an open application programming interface (API) for CANopen slave nodes. All CANopen solutions providing this API are interchangeable and allow the developer of a CANopen node to switch between different implementations with different functionality or performance. A CANopen design started based on a basic or minimal CANopen implementation could still be upgraded later.

One of the main goals of the API is to hide as many CANopen specific details from the host (application process) as possible. The host only worries about the process data it exchanges with the network. When and how the data is transmitted is entirely left to the CANopen implementation and its configuration.

1.1 Definition of Terms

This document uses many CANopen specific terms and abbreviations. All CANopen terms used are explained in the CANopen standard as published by the CiA (CAN in Automation manufacturer's group) in document DS301.
In addition, the following terms and abbreviations are used in this document:

ACK
Positive acknowledgement used on a message oriented interface between the Host and the CANopen Task. The previously sent request or command executed successfully.

BYTE
All parameters, values and numbers in this document are BYTE oriented. In C programming language, a BYTE is a "unsigned char".

CANopen API
The CANopen API is the application programming interface used between the Host and the CANopen Task.

CANopen Task
The CANopen Task is the part of a CANopen implementation that executes all functions related to the CANopen protocol. The CANopen Task can either be executed on the same microcontroller as the Host or it can be running on an additional microcontroller implementing a CANopen coprocessor.

FALSE
A function that returns FALSE returns the value 00h.

Host
The Host is the part of a CANopen implementation that executes all functions related to processing the application. The code implementing the Host functionality can either be executed on the same microcontroller that is running the code for the CANopen Task or it can be executed on a separate microcontroller if the CANopen Task runs on its own, dedicated microcontroller.

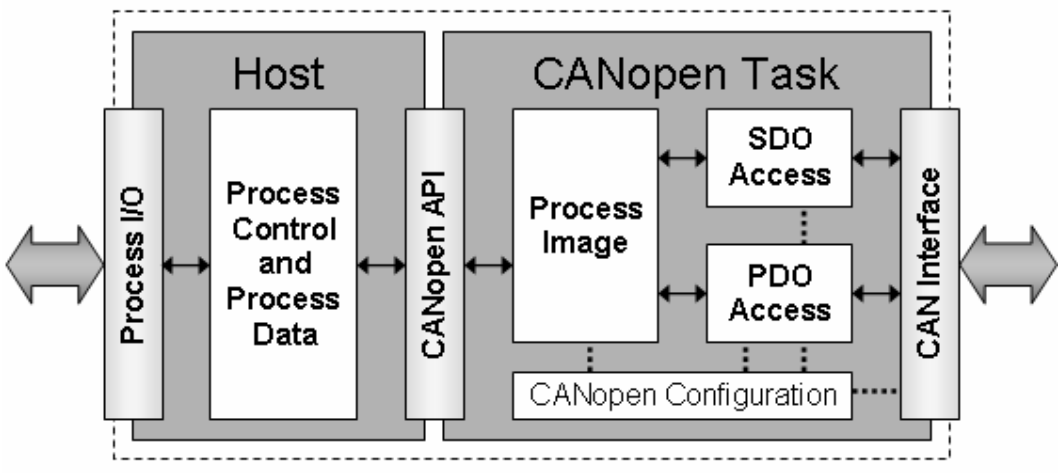
NAK
Negative acknowledgement used on a message oriented interface between the Host and the CANopen Task. The previously sent request or command did not execute successfully.

Process Image
The process data that can be exchanged between the Host and the CANopen Task is organized into a process image, an array of bytes.

TRUE
A function that returns TRUE returns the value FFh.

1.2 Block Diagram

The block diagram below shows the main function blocks of a CANopen node based on the CANopen API. The CANopen Task implements all the basic CANopen features and functions required by the node. It also holds all configuration data like which Object Dictionary entries are stored where in the Process Image and which bytes from the Process Image are mapped into which PDOs (Process Data Objects).



The Host implements the main process control application and directly handles all inputs and outputs to and from the application process. Using the CANopen API, the Host can exchange selected process data with the CANopen Task which is stored into the Process Image.

The Host and the CANopen Task can either be implemented on a single microcontroller running as separate tasks or functions or they can be implemented on two dedicated microcontrollers using a message oriented communication channel (for example shared memory with mail boxes or a serial channel) between them.

2 Interface Implementation Options

CANopen implementations are available in a wide range of product types. There are software solutions in form of a code library or source code and there are hardware solutions implementing CANopen peripheral coprocessors on chips or modules.

The CANopen API introduced in this document is language and communication channel independent. The commands and responses defined can be used with any programming language as well as on communication channels, such as a serial channel between the application and the CANopen implementation.

2.1 Interfacing with Programming Languages

When the interface is a programming language, then a set of functions are used by the Host to send requests to the CANopen Task. All functions have a return value so that the Host gets immediate feedback if the request was executed successfully or not.

Indications from the CANopen Task back to the host are sent via call-back functions. These are functions that must be implemented within the Host program, as they will be called by the CANopen Task

2.2 Interfacing with a Message System

When a message system is used as a communication channel between the Host and the CANopen Task, then each function is identified with a function code.

The end of every message sent from the Host to the CANopen Task is marked with a semi-colon ";". The end of every message or response sent from the CANopen Task to the Host is marked with a dot "." or an exclamation mark "!". The "." is used as an acknowledgement (ACK) in response to a request or to mark the end of an indication. The "!" is used as a negative acknowledge (NAK) to inform the host that the last request could not be executed.

3 Using a Process Image

The host of a device with CANopen network access primarily deals with process data. It has a set of input variables that come from the application and are transmitted into the network and/or it has a set of output variables that are received from the network and applied to the application.

3.1 The CANopen API Process Image

In order to offer a generic method for addressing and exchanging the process data, the process data is organized into a process image which is implemented as an array of bytes with a maximum length of 255.

A single variable of the process image can be addressed by specifying an offset and a length. The offset specifies where in the process image the first byte of a variable is stored and the length specifies how many bytes are used to store the variable. The offset may have a value from 0 to 254. Using an offset of 255 indicates that the offset is invalid or unused.

If numeric values are stored in multiple byte variables, then the format is CANopen compatible: Little Endian – the lower bytes are stored at the lower offset.

3.2 Configuring the Usage of the Process Image

Where exactly which variable is located in the process image is part of the CANopen node configuration process. Whichever tool is used to configure the CANopen part of the node also should configure the usage of the process image.

The CANopen configuration process also includes assigning an Object Dictionary Index and Subindex to each variable and to configure the PDOs (Process Data Objects) containing one or multiple process data variables. For more details on the configuration of the process image, see chapter 5.

3.3 Accessing the Process Image

To provide data consistency in the process image the data in it should never be used directly (like a host writing directly to the process image). Instead, the functions specified by this document should be used. Ensuring data consistency is the responsibility of these access functions.

4 CANopen API Functions

4.1 Requests from Host to CANopen Task

This section lists all the commands/requests send from the Host to the CANopen Task. Examples are given for both an implementation using C function calls and using a message system (for example a serial channel).

4.1.1 Function GetID

This function allows the Host to request identification information from the CANopen Task.

Name	GetID
Code	'V', 53h
Parameter	None
Returns	The 4 identification bytes of the CANopen task including a version number.

The 4 identification bytes are configurable and can be used to identify specific implementations or configurations of the CANopen Task. The first two bytes contain a manufacturer specific identification for this particular CANopen Task (first byte contains low-byte and second byte contains high-byte). Values from F000h to FFFFh are reserved and should not be used. The remaining two bytes contain a version number with the third byte being a minor version number and the fourth byte the major version number.

Typically these 4 identification bytes are configurable. See chapter 5 for more details on the configuration of the CANopen Task.

C-Example:

```
BYTE[4] COAPI_GetID
     ( void );
```

Message-Example:

Request: `V;`
ACK: `<identification bytes>.`

4.1.2 Function GetStatus

This function allows the Host to check upon the general status of the CANopen Task.

Name	GetStatus
Code	'S', 53h
Parameter	None
Returns	The status byte of the CANopen task

The following values are defined for the status byte of the CANopen Task:

Status	Description
Bit 0-1	Basic Status of the CANopen Task: 00b – Not yet ready to operate, still initializing 01b – Ready to operate, waiting for Host to send the InitCANopen command 10b – CANopen Task is initialized and operating 11b – Reserved
Bit 2-4	Size of the Process Image: (value + 1) * 32 Exception: In case of 111b the size is 255
Bit 5-7	Reserved

C-Example:

```
BYTE COAPI_GetStatus
     ( void );
```

Message-Example:

Request: `S;`

ACK: `<status byte>.`

4.1.3 Function InitCANopen

The Host must call this function to initialize the CANopen Task. Without this call the CANopen Task does not start to operate. Repetitious calls cause the CANopen Task to re-initialize.

Name	InitCANopen
Code	'I', 49h
Parameter	`bps` – The CANopen bit rate used `nodeid` – The CANopen node ID used
Returns	TRUE or ACK, if initialization was successful FALSE or NAK , if initialization was not successful

The "`bps`" value must be in the range of 1 to 8 representing one of the bit rates listed below. If "`bps`" has a value of zero, the CANopen Task will use its internal default setting for the bit rate.

Baudrate	Bps Value
1 MBit/s	8
800 kBit/s	7
500 kBit/s	6
250 kBit/s	5
125 kBit/s	4
50 kBit/s	3
20 kBit/s	2
10 kBit/s	1

The value "`nodeid`" selects the CANopen node ID used by this node and must be in the range of 1 to 127. If "`nodeid`" is zero, the CANopen Task will use its default node ID number.

C-Example:

```
BYTE COAPI_InitCANopen
     ( BYTE bps, BYTE nodeid );
```

Message-Example:

Request: `I<bps><nodeid>;`
ACK: `.`
NAK: `!`

4.1.4 Function ReadProcessData

This function is used by the Host to read data from the Process Image.

Name	ReadProcessData
Code	'R', 52h
Parameter	`offset` – Offset to first requested byte from Process Image `len` – Number of bytes requested `pDat` – Destination pointer to where requested data should be copied
Returns	Number of bytes read from Process Image. Zero if no bytes were returned (due to illegal values for offset and len)

The parameter "`offset`" selects the offset the first byte requested has in the Process Image. The parameter "`len`" specifies how many bytes starting at "offset" will be copied to the location set by the pointer "`pDat`".

The return value indicates the number of data bytes actually copied. It will be zero if "`offset`" is bigger than the Process Image or might be a value smaller than "`len`" if the end of the Process Image is reached with the read request.

C-Example:

```
BYTE COAPI_ReadProcessData
     ( BYTE offset, BYTE len, BYTE *pdata );
```

Message-Example:

Request: `R<offset><len>;`
ACK: `<len><data bytes>.`
NAK: `0!`

4.1.5 Function WriteProcessData

This function is used by the Host to write data to the Process Image.

Name	WriteProcessData
Code	'W', 57h
Parameter	`offset` – Offset to first destination byte in Process Image `len` – Number of bytes to be written `pDat` – Source pointer to the data copied to the Process Image
Returns	Number of bytes written to Process Image. Zero if no bytes were written (due to illegal values for offset and len)

The parameter "`offset`" selects the offset the first byte written to has in the Process Image. The parameter "`len`" specifies how many bytes starting at "offset" will be copied into the Process Image from the source indicated by the pointer "`pDat`".

The return value indicates the number of data bytes actually copied. It will be zero if "`offset`" is bigger than the Process Image or might be a value smaller than "`len`" if the end of the Process Image is reached with the write request.

C-Example:

```
BYTE COAPI_WriteProcessData
     ( BYTE offset, BYTE len, BYTE *pdata );
```

Message-Example:

Request: `W<offset><len><data>;`
ACK: `<len>.`
NAK: `0!`

4.2 Indications from CANopen Task to Host

This section lists all the indications from the CANopen Task to the Host. Examples are given for both an implementation using C call-back functions (called from CANopen API, must be implemented by Host) and using a message system (for example a serial channel).

4.2.1 Callback Function NMTChange

This indicates that the CANopen Task changed its CANopen Network Management state or received a request to reset itself.

Name	NMTChange
Code	'n', 6Eh
Parameter	`nmtstate` – The current NMT state of the CANopen node

The value "`nmtstate`" can have one of the following values:

nmtstate	Description
00h	Initializing (sent after receiving the 'I' command)
04h	CANopen NMT state "stopped" entered
05h	CANopen NMT state "operational" entered
7Fh	CANopen NMT state "pre-operational" entered
81h	The CANopen Task received an NMT request to reset the entire CANopen node, the Host should reset itself
82h	The CANopen Task received an NMT request to reset the CAN communication interface
other	All other values are reserved

C-Example:

```
void COAPICB_NMTChange
     ( BYTE nmtstate );
```

Message-Example:

Sent by Task: `n<nmtstate>.`

4.2.2 Callback Function ReceivedData

This is an indication from the CANopen Task that new data arrived and was copied into the Process Image. To read the data, the Host needs to call the function ReadProcessData.

Name	ReceivedData
Code	'd', 64h
Parameter	`offset` – Offset to first byte in Process Image that was received from CANopen `len` – Number of bytes that were received

The parameter "`offset`" specifies the location of the first byte in the process image that was updated via CANopen and "`len`" indicates the number of bytes that were updated.

C-Example:

```
void COAPICB_ReceivedData
     ( BYTE offset, BYTE len );
```

Message-Example:

Sent by Task: `d<offset><len>.`

NOTE:
If a RPDO contains data that is copied to consecutive entries in the process image, only one indication will be issued by the CANopen Task. If due to the RPDO mapping the data goes to non-consecutive locations in the process image, then multiple indications will be issued. It is recommended to ensure that data belonging to the same RPDO is stored in consecutive locations in the process image.

4.2.3 Callback Function FatalError

This indication signals the Host that the CANopen Task ran into a fatal error situation and needs to be reset or re-initialized to start operation again.

Name	FatalError
Code	'f', 66h
Parameter	`errorcode` – an internal error code

C-Example:

```
void COAPICB_FatalError
     ( BYTE errorcode );
```

Message-Example:

Sent by Task: `f<error code>.`

5 CANopen Task Setup File

Although working with CANopen EDS and DCF files is the standard procedure for many CANopen configuration tools, these file formats are not very suitable to be handled by embedded microcontrollers. This document recommends the usage of an ASCII file optimized towards the setup of the CANopen Task.

5.1 Generic Setup File Format

The setup file is an ASCII text file with the default ending of ".txt" to allow simple editing with any text editors.

The file content is organized by lines. Lines starting with a semi-colon ";" or a slash "/" are regarded comment and are ignored. So are all spaces, line-feed and return characters.

All data values are in hexadecimal, using capitalized letters for the letters "A" through "F". Additional characters like "0x" or "h" are not allowed. If multiple bytes are used for a data value (for example WORD or DWORD), the byte ordering is Little Endian (lower significant byte(s) first).

The first line MUST start with the 8 characters "COPTSKA8". Additional characters may be used to indicate a version number, for example "COPTSKA8 V1.0"

It is recommended to make the second line a comment line with details about the purpose of this configuration and a generation timestamp, for example "// For MyEncoder, generated on 18-AUG-03 by John Doe"

5.2 Entries in the Setup File

Each entry section starts with a line containing a label in square brackets. All labels use upper case letters only, for example: "[NODE]". This section contains a list of all labels defined. Note that the label length is kept short, to simplify processing for embedded microcontrollers.

The line or lines following a label contain the data values for that entry.

The last line of an entry contains a single byte, the checksum for this entry. The checksum byte is calculated by adding up all bytes of the entry.

5.2.1 Entry [ID]

This is the 4-byte identification value reported by the CANopen Task when the Host uses the "GetID" function (see section 4.1.1).

The first two bytes contain a manufacturer specific identification for this particular CANopen Task (first byte contains low-byte and second byte contains high-byte). Values from F000h to FFFFh are reserved and should not be used. The remaining two bytes contain a version number with the third byte being a minor version number and the fourth byte the major version number.

5.2.2 Entry [NODE]

This data field contains a data record with the basic setup information for the CANopen Task.

Byte Nr.	Name	Description
1	Bps	Default CAN bit rate. Same contents as used for the function InitCANopen (see section 4.1.3)
2	Node ID	Default CANopen bit rate. Same contents as used for the function InitCANopen (see section 4.1.3)
3	Process Image Size	The size of the process image used by the CANopen Task. Maximum size allowed is 255.
4	Nr of RPDOs	The number of RPDOs used by the CANopen Task.
5	Nr of TPDOs	The number of TPDOs used by the CANopen Task.
6	Functionality	Each bit in this entry can disable/enable a certain CANopen functionality in the CANopen Task, such as allowing dynamic PDO mapping or not. Usage is manufacturer specific.

Example:

```
[NODE]
044020020200
68
```

The example selects a default bit rate of 125kbps, a default node ID of 40h (64d), a process image size of 20h (32d), 2 RPDOs and 2 TPDOs. The checksum for this entry is 68h.

5.2.3 Entry [RPDO]

This data entry contains the communication and mapping parameters for each RPDO used. The number of data fields must be matching the Nr of RPDOs value used in the entry [NODE].

The first data field contains 3 bytes for each RPDO holding the communication parameters. The first 2 bytes set the COB-ID and the 3rd byte the transmission type.

Byte Nr.	Name	Description
1-2	COB-ID	RPDO COB-ID, leave at zero to use default
2	Transmission Type	RPDO transmission type, typically FEh or FFh

The second data field contains 9 bytes for each RPDO holding the mapping parameters. The first byte of each entry specifies the number of entries mapped. Allowed values are 0 through 8. The following entries identify the Object Dictionary entry mapped using a single byte. The byte is the offset that the mapped Object Dictionary entry has in the [RWOD] section. Unused bytes must be set to FFh.

Byte Nr.	Name	Description
1	Nr of Entries	Number of mapping entries for this RPDO
2-9	Mapping	Mapping entries, each byte refers to an entry (starting at 0) in the list of Object Dictionary entries [RWOD]

Example:

```
[RPDO]
0000FF
0000FF
0400010203FFFFFFFF
020405FFFFFFFFFFFF
09
```

The example is for Nr of RPDOs being two. The COB-IDs selected are zero, meaning the CANopen default COB-IDs (from the pre-defined connection set) should be used. The transmission type is FFh.

The first RPDO has 4 Object Dictionary entries mapped. The mapped entries are the first 4 Object Dictionary entries listed in the [RWOD] section. The second

RPDO has 2 entries mapped. They are the 5th and 6th entries listed in [RWOD]. The checksum for this example is 09h.

5.2.4 Entry [TPDO]

This entry contains the communication and mapping parameters for each TPDO used. The number of data fields must be matching Nr of TPDOs value used in the entry [NODE].

The first data field contains 7 bytes for each TPDO holding the communication parameters. The first 2 bytes set the COB-ID, the next 2 bytes the inhibit time, the next 2 bytes the event time and the 7th byte the transmission type.

Byte Nr.	Name	Description
1-2	COB-ID	TPDO COB-ID, leave at zero to use default
3-4	Inhibit Time	The TPDO Inhibit Time in 100s of microseconds
5-6	Event Time	The TPDO Event Time in milliseconds
7	Transmission Type	TPDO transmission type, typically FEh or FFh

The second data field contains 9 bytes for each TPDO holding the mapping parameters. The first byte of each entry specifies the number of entries mapped. Allowed values are 0 through 8. The following entries identify the Object Dictionary entry mapped using a single byte. The byte is the offset that the mapped Object Dictionary entry has in the [RWOD] section. Unused bytes must be set to FFh.

Byte Nr.	Name	Description
1	Nr of Entries	Number of mapping entries for this TPDO
2-9	Mapping	Mapping entries, each byte refers to an entry (starting at 0) in the list of Object Dictionary entries [RWOD]

Example:

```
[TPDO]
0000F401FA00FF
0000C800E803FF
020607FFFFFFFFFFFF
020809FFFFFFFFFFFF
B6
```

The example is for Nr of TPDOs being two. The COB-IDs selected are zero, meaning the CANopen default COB-IDs (from the pre-defined connection set) should be used. The inhibit times are 01F4h (500d) for the first and 00C8h (200d) for the second TPDO. The event times are 00FA (250d) for the first and 03E8 (1000d) for the second TPDO.

Both TPDOs have 2 Object Dictionary entries mapped. The mapped entries for the first TPDO are the 7^{th} and 8^{th} Object Dictionary entries listed in the [RWOD] section. The second TPDO use the 9^{th} and 10^{th} entries listed in [RWOD]. The checksum for this example is B6h.

5.2.5 Entry [COD]

This entry contains a list of SDO responses for SDO requests to constant, read-only entries in the object dictionary. Typically these contain the [1000,00] Device Type entry, the [1018,xx] Identity Objects and the "Number of Entries" type entries with a Subindex of zero.

Each entry in this list has 8 bytes that directly contain the 8 bytes used in a CAN message with an expedited SDO response to a read (upload) request.

Byte Nr.	Name	Description
1	CS	SDO response command specifier
2-3	Index	Index of the Object Dictionary entry
4	Subindex	Subindex of the Object Dictionary entry
5-7	Data	Data bytes to be send in the response

The last entry in this list must consist of 8 bytes with the value FFh.

Example:

```
[COD]
4300100091010F00
430810004C585858
4F18100003000000
4318100141534501
431810024C58794D
4318100350000100
4F00600006000000
4F00620004000000
FFFFFFFFFFFFFFFF
xx
```

The example contains the SDO responses for the following Object Dictionary entries:
[1000,00]: returns 000F0191h
[1008,00]: returns 5858584Ch (“XXXL”)
[1018,00]: returns 03h
[1018,01]: returns 01455341h
[1018,02]: returns 4D79584Ch
[1018,03]: returns 00010005h
[6000,00]: returns 06h
[6200,00]: returns 04h

5.2.6 Entry [RWOD]

This entry contains the list of Object Dictionary entries that address data in the process image. Each entry in this list has 5 bytes.

Byte Nr.	Name	Description
1-2	Index	Index of the Object Dictionary entry
3	Subindex	Subindex of the Object Dictionary entry
4	Access and length	This byte contains the access-type and length information for this Object Dictionary entry, for details see next table.
5	Offset	Offset to a location in the process image where the data for this Object Dictionary entry is stored.

The first 2 bytes specify the Index and the 3rd byte the Subindex of the Object Dictionary entry. The 4th byte contains the length information combined with access type bits. The bits in this byte are used as follows:

Bit	Description
Bit 0-2	Length of the data in this Object Dictionary entry. Must be in the range of 1 through 4
Bit 3	Reserved
Bit 4	If set, SDO read (upload) access is allowed
Bit 5	If set, SDO write (download) access is allowed
Bit 6	If set, this entry can be mapped to a PDO
Bit 7	If bit 6 is set, this bit specifies the direction of the mapping: If 0, the entry can be mapped to a TPDO only. If 1, the entry can be mapped to a RPDO only.

The 5th byte indicated the offset to the data in the process image that belongs to this Object Dictionary entry.

The last entry in this list must consist of 5 bytes with the value FFh.

Example:

```
[RWOD]
0060015100
0060025101
0060035102
0060045103
0060055104
0060065105
006201F106
006202F108
006203F108
006204F109
FFFFFFFFFF
xx
```

The Object Dictionary entries specified by this example are:
[6000,01]: read-only, TPDO mapping, 1 byte of process image at offset 0
[6000,02]: read-only, TPDO mapping, 1 byte of process image at offset 1
through
[6000,06]: read-only, TPDO mapping, 1 byte of process image at offset 5
[6200,01]: read-write, RPDO mapping, 1 byte of process image at offset 6
through
[6200,04]: read-write, RPDO mapping, 1 byte of process image at offset 9

5.2.7 Entry [PIMG]

This entry contains the default data for the process image. During initialization of the CANopen Task, this data will be copied to the process image. The length of this data field must be identical to the length of the process image specified in the [NODE] entry.

Example:

```
[PIMG]
0011223344556677
8899AABBCCDDEEFF
0011223344556677
8899AABBCCDDEEFF
xx
```