AN11238

LPC11Cxx CANopen network demo

Rev. 1 — 12 July 2012 Application note

Document information

Info	Content
Keywords	LPC11C12FBD48; LPC11C14FBD48; LPC11C22FBD48; LPC11C24FBD48, LPC11Cxx, CAN, CANopen, on-chip, driver
Abstract	This application note illustrates how to use LPC11Cxx to setup a simple CANopen network using the on-chip CAN/CANopen drivers.

Revision history

Rev	Date	Description
1	20120712	Initial release.

Contact information

For additional information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

This application note demonstrates the CAN peripheral and the on-chip CAN/CANopen drivers of the LPC11Cxx.

1.1 CAN

The Controller Area Network (CAN) is a serial, asynchronous, multi-master communication protocol for connecting electronic control modules, sensors and actuators in automotive and industrial applications. Important features of CAN are:

- Very good signal integrity due to differential signaling.
- Collision and error detection and re-transmission possibilities.
- Multiple masters can exist within one network, allowing intelligent and redundant systems.

A CAN frame exists of 11 identifier bits and 8 data bytes. CAN does not specify the meaning of these identifier bits, nor the meaning of the data bytes. A high-level protocol is necessary for this.

1.2 CANopen

CANopen is a type of high-level protocol for CAN bus**,** suitable for embedded networks. It is built on top of CAL, using a subset of CAL services and communication protocols. It provides an implementation of a distributed control system using the services and protocols of CAL.

The central concept in CANopen is the device Object Dictionary (OD). For every node in the network there exists an OD. The OD contains all parameters describing the device and its network behavior. The OD is an ordered grouping of objects; each object is addressed using a 16-bit index. To allow individual elements of structures of data to be accessed, an 8-bit sub-index has been defined as well. Fig 1 shows the general layout of CANopen OD.

CANopen Object Dictionary	
Index	**Object**
0000	*not used*
0001 - 001F	Static Data Types (standard data types, e.g. Boolean, Integer16)
0020 - 003F	Complex Data Types (predefined structures composed of standard data types, e.g. PDOCommPar, SDOParameter)
0040 - 005F	Manufacturer Specific Complex Data Types
0060 - 007F	Device Profile Specific Static Data Types
0080 - 009F	Device Profile Specific Complex Data Types
00A0 - 0FFF	*reserved*
1000 - 1FFF	Communication Profile Area (e.g. Device Type, Error Register, Number of PDOs supported)
2000 - 5FFF	Manufacturer Specific Profile Area
6000 - 9FFF	Standardised Device Profile Area (e.g. "DSP-401 Device Profile for I/0 Modules" [3]: Read State 8 Input Lines, etc.)
A000 - FFFF	*reserved*

Fig 1. General CANopen Object Dictionary structure

The CANopen communication model defines four types of communication objects (COB):

- Administrative message.
- Service Data Object (SDO).
- Process Data Object (PDO).
- Predefined messages or Special Function Objects.

An SDO provides the access to entries of a device OD. PDO is used to transfer real-time data. Predefined messages or special function objects include synchronization, time stamp, emergency, node/life guarding and boot-up.

The relation between CAN communication, the Object Dictionary and application software on a CANopen device is schematically illustrated in Fig 2.

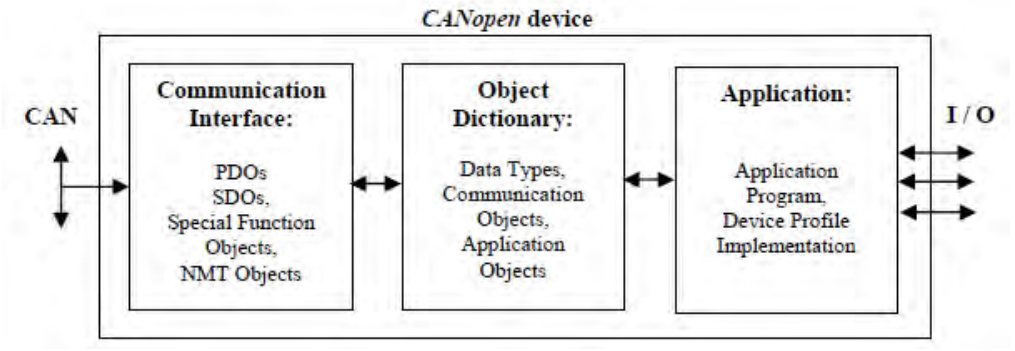

Fig 2. Schematic relationship between CAN-bus communication, Object Dictionary and application software on a CANopen device

Different CAN identifiers support different communication objects, called COB-ID. The predefined connection set defines PDO, SDO, emergency object and node error-control identifier. It also supports the broadcasting of NMT module control, SYNC and time stamp objects. Fig 3 shows the CAN-identifier allocation scheme.

Broadcast objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID**	**Communication parameters at OD index**
NMT Module Control	0000	000h	–
SYNC	0001	080h	1005h, 1006h, 1007h
TIME STAMP	0010	100h	1012h, 1013h

Peer-to-Peer objects of the CANopen Predefined Master/Slave Connection Set			
Object	**Function code (ID-bits 10-7)**	**COB-ID ***	**Communication parameters at OD index**
EMERGENCY	0001	081h - 0FFh	1024h, 1015h
PDO 1 (transmit)	0011	181h - 1FFh	1800h
PDO 1 (receive)	0100	201h - 27Fh	1400h
PDO 2 (transmit)	0101	281h - 2FFh	1801h
PDO 2 (receive)	0110	301h - 37Fh	1401h
PDO 3 (transmit)	0111	381h - 3FFh	1802h
PDO 3 (receive)	1000	401h - 47Fh	1402h
PDO 4 (transmit)	1001	481h - 4FFh	1803h
PDO 4 (receive)	1010	501h - 57Fh	1403h
SDO (transmit/server)	1011	581h - 5FFh	1200h
SDO (receive/client)	1100	601h - 67Fh	1200h
NMT Error Control	1110	701h - 77Fh	1016h, 1017h

Fig 3. CANopen predefined CAN-identifiers

For CANopen protocol details, please refer to “CANopen Application Layer and Communication Profile” (CiA DS301).

1.3 LPC11Cxx

Built around the Cortex-M0 architecture, LPC11Cxx is integrated with one C_CAN controller, offering a low-cost entry point for CAN-based applications and on-chip CAN/CANopen drivers.

The LPC11Cxx has one C_CAN controller. The C_CAN controller is designed to provide a full implementation of the CAN protocol according to the CAN Specification Version 2.0B.

The main features of the LPC11Cxx CAN peripheral:

- Conforms to protocol version 2.0 parts A and B.
- Supports bit rate of up to 1 Mbit/s.
- Supports 32 Message Objects.
- Each Message Object has its own identifier mask.
- Provides programmable FIFO mode (concatenation of Message Objects).
- Provides maskable interrupts.
- Supports Disabled Automatic Retransmission (DAR) mode for time-triggered CAN applications.
- Provides programmable loop-back mode for self-test operation.
- On-chip CAN/CANopen drivers.

The CAN controller consists of a CAN core, message RAM, a message handler, control registers, and the APB interface. For communication on a CAN network, individual

Message Objects are configured. The Message Objects and Identifier Masks for acceptance filtering of received messages are stored in the Message RAM.

All functions concerning the handling of messages are implemented in the Message Handler. Those functions are the acceptance filtering, the transfer of messages between the CAN Core and the Message RAM, and the handling of transmission requests as well as the generation of the module interrupt.

The register set of the CAN controller can be accessed directly by an external CPU via the APB bus. These registers are used to control/configure the CAN Core and the Message Handler and to access the Message RAM.

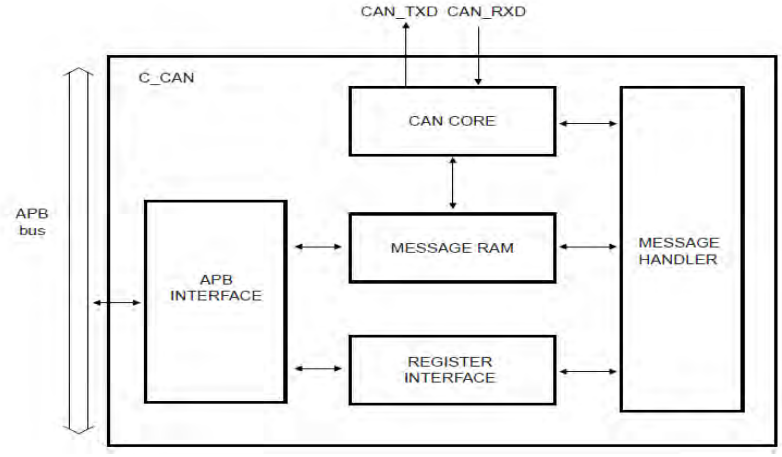

Fig 4. C_CAN block diagram

2. CANopen demo introduction

The demo requires a minimum of two LPC11Cxx boards. Three types of LPC11Cxx boards are supported:

- Keil MCB11C14 (full support).
- IAR LPC11C14-SK (not all demo functions are supported).
- LPCXpresso LPC11C24 (not all demo functions are supported).

Additionally, a third LPC11Cxx board can be connected to create a larger network.

Three IDE's are supported:

- Code Red LPCXpresso, tested with V4.1.5.
- Keil µVision, tested with V4.23.
- IAR EWARM, tested with V6.30.

The master node requires about 22 kB of flash and 1.5 kB of RAM.

The slave node only requires about 5 kB of flash and 1.5 kB of RAM.

The big difference between the two is because the master node application uses the full printf() and scanf() functions. The master node and the slave node share the same CANopen library, which is quite small.

For a fully functional demo, the following setup is required (Fig 5):

- One MCB11C14 board flashed with the master software.
- Two MCB11C14 boards flashed with slave software (both boards should have unique CAN_NODE_ID).
- Master board must be connected to a computer's serial port, running a terminal program (e.g. PuTTY).
- To see what data is transmitted over the CAN bus, a CAN bus analyzer is highly recommended.

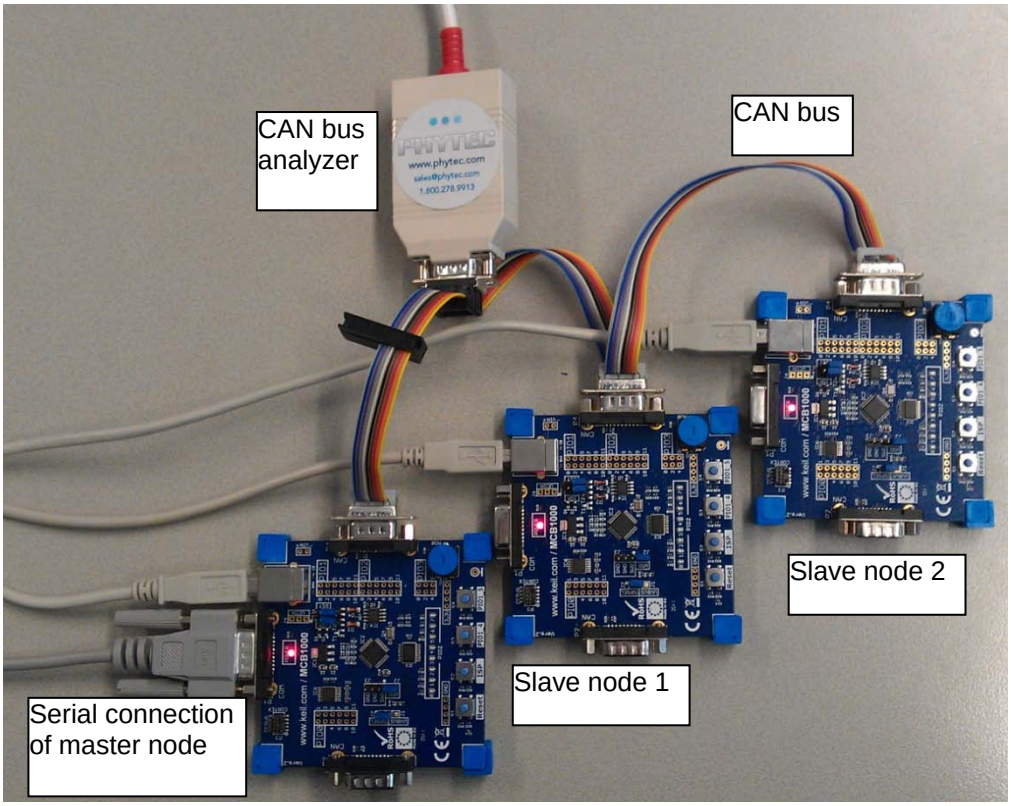

Fig 5. Demo setup with three Keil MCB11C14 boards. The USB cables are only used for supplying power to the boards

Not all features of CANopen are implemented in the demo, but the most-used features of CANopen are supported:

- SDO expedited transfer: client & server.
- SDO segmented transfer: client & server.
- NMT: master, slave & boot-up.
- Heartbeat: producer & consumer.

The demo will demonstrate these features with the aid of a terminal program and the onboard LEDs and buttons.

PDO transfer, synchronization, time stamp and emergency are not realized in the demo.

The LPC11Cxx also provides on-chip C_CAN drivers. In addition to the CAN ISP, the boot ROM provides a CAN and CANopen API to simplify CAN application development. It covers initialization, configuration, basic CAN send/receive as well as a CANopen SDO interface. Callback functions are available to process received events. This application makes use of these on-chip CAN/CANopen drivers, thereby reducing development time and saving FLASH memory. However, the on-chip CANopen driver does not fully implement all CANopen functions which are used by this demo, so for the demo application an additional driver has been created. This driver is generic for all CANopen functions, and thus is shared among the master and slave software.

3. Running the demo

3.1 Introduction

As stated in chapter 2, three types of LPC11Cxx boards are supported. The demo consists of several small programs which can be run in two ways:

- Some functions of the demo rely on the onboard LEDs and buttons of the Keil MCB11Cxx board.
- Some functions of the demo rely on the serial connection of the master node (to a terminal program like PuTTY).

Note: Though multiple boards are supported, the MCB11Cxx is the only board on which the demo can be partially run without serial connection. When using a board other than the MCB11Cxx, there is no way of controlling the demo other than by the serial connection.

3.2 Using the RS232 serial connection

The demo board can be controlled by a RS232 serial connecting by starting a terminal application with the following settings:

- Baudrate: 115200.
- Databits: 8.
- Parity: none.
- Stopbits: 1.

After booting up, the master node prints a menu to the terminal, which can be seen in Fig 6.

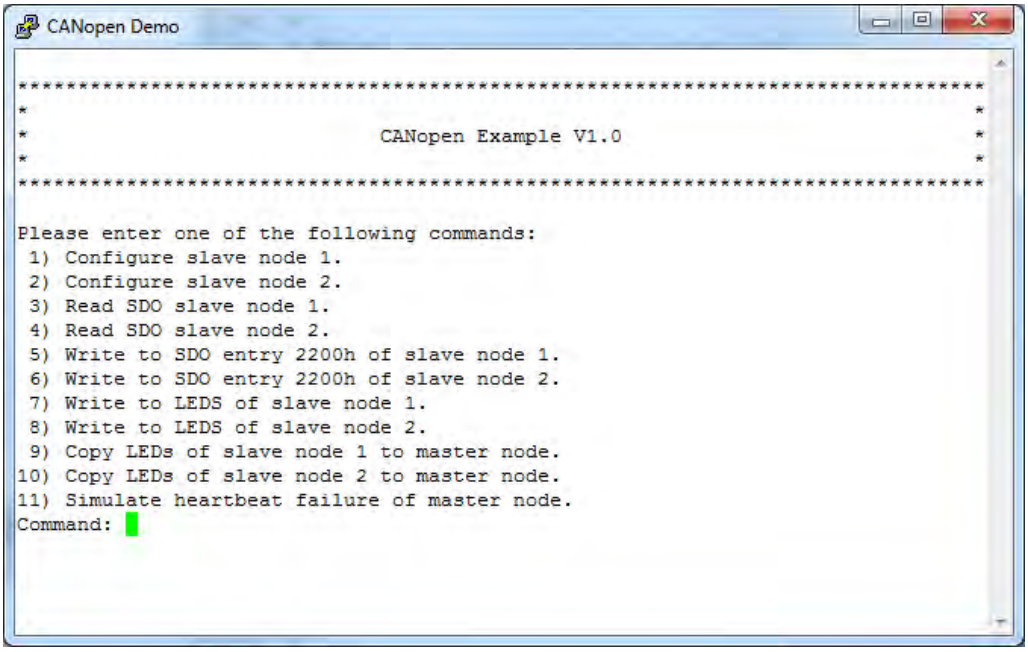

Fig 6. On startup of the CANopen master node, a menu is printed to the terminal

By entering any of the command numbers (1-11), an action is performed. Short summary of these actions are as given below:

1 & 2) Sets the heartbeat producer time of the specified slave to 4000 ms, its heartbeat consumer time to 4500 ms and the heartbeat consumer node ID to the master's node ID.

This configures the slave to generate a heartbeat every 4000 ms and to track the heartbeat of the master node. If the master does not send a heartbeat within every 4500 ms, the slave node tries to reset the master node.

3 & 4) Read the full SDO of the specified slave node and prints it to the terminal. Fig 7 shows the text from the terminal when reading the SDO of slave node 1 after configuring it.

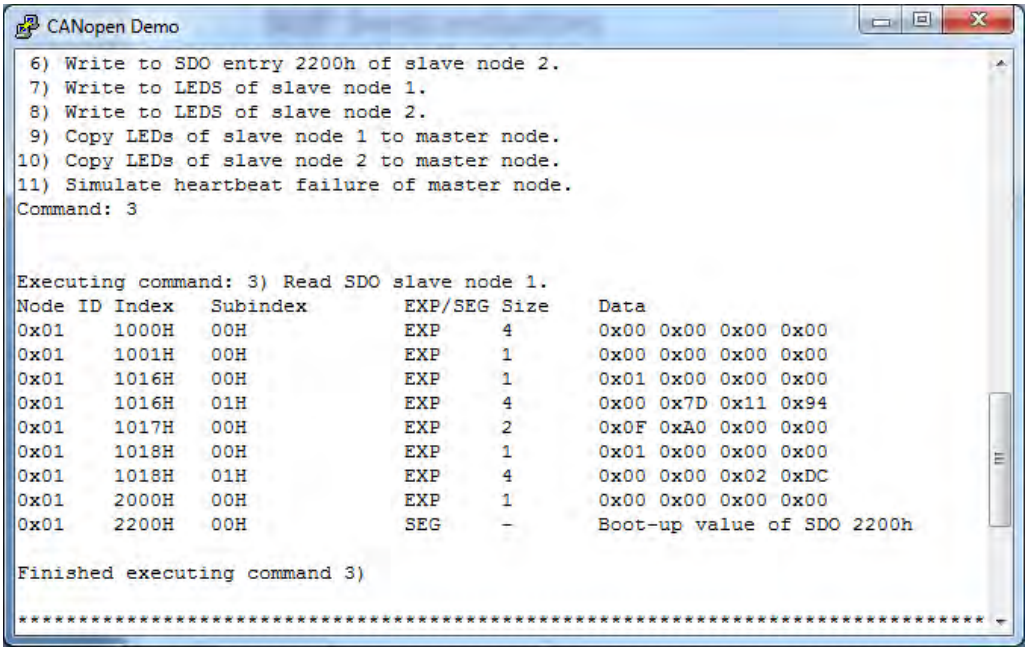

Fig 7. Result of execution of command 3)

5 & 6) After selecting command (5 or 6), the user is asked to enter a string. On pressing enter, the string is sent to SDO entry 2200H of the specified slave. The result can be verified by re-reading the complete SDO of the specified slave. An example is given in Fig 8.

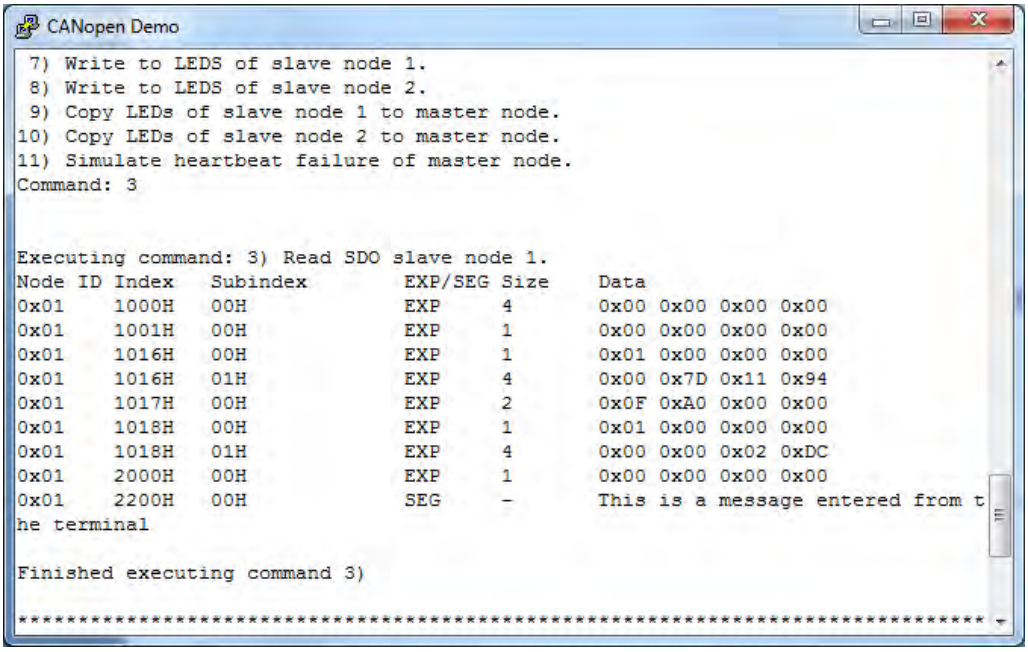

Fig 8. Result of execution of command 3) after previously changing SDO entry 2200H by command 5)

7 & 8) By writing to SDO entry 2000H of the slave nodes, their LEDs (GPIO2) can be controlled. After selecting command 7 or 8, the user is asked to enter a hexadecimal value. After entering the same and pressing enter, the hexadecimal value is sent to the SDO entry 2000H of the specified slave node. The slave node copies the value of SDO entry 2000H to the LEDs (GPIO2) every 1 ms. Reading the SDO of the specified slave shows the written hexadecimal value in SDO entry 2000H.

9 & 10) Reads SDO entry 2000H of specified slave and copies this value to the LEDs (GPIO2) of the master node.

11) On start-up of the master node, it immediately starts sending its heartbeat every 4000 ms. By executing command 11), the master node stops sending its heartbeat, thereby simulating a failing master node.

If one or more slave nodes have been configured by command 1) and 2), the configured slave node(s) will send the NMT command “Reset Communication” after detection of the missing heartbeat. Fig 9 shows the output of the terminal after the NMT command has been received.

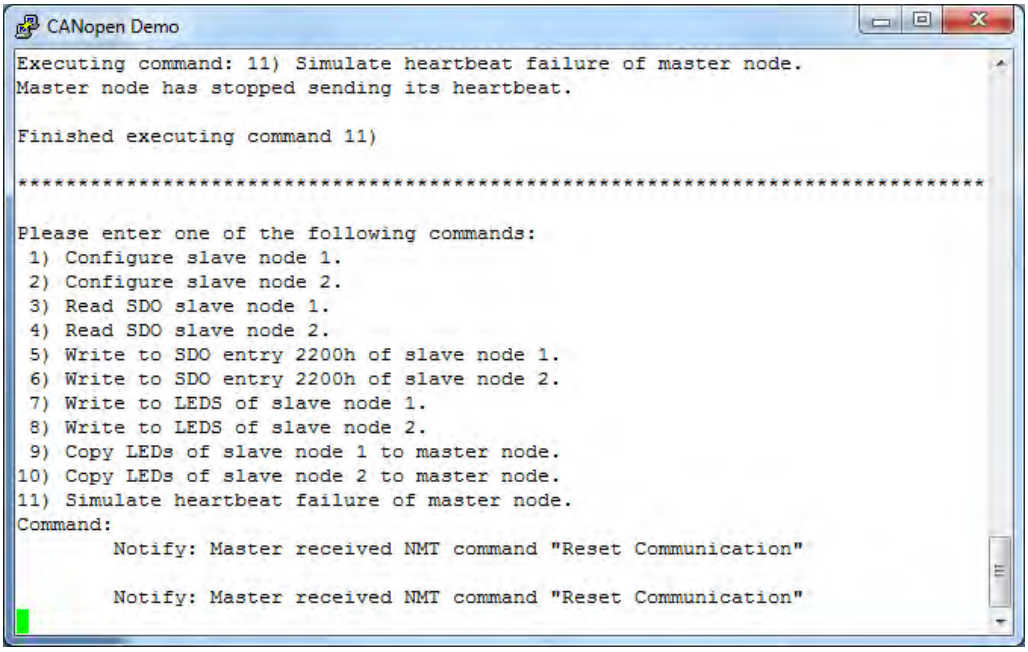

Fig 9. Result of execution of command 11). Master received NMT command “Reset Communication” from both slaves.

3.3 Using the onboard buttons

The demo can also be partially controlled by the onboard buttons. Please see Table 1 for details of the functions that can be executed by the buttons.

Table 1. The demo can also be controlled by the onboard buttons. This table shows how the functions are mapped to the buttons

Button	Master board function	Slave board function
ISP	Simulate heartbeat failure of the master node.	Simulate heartbeat failure of the slave node.
PIO1_4	Read SDO entry 2000H (LED value) of slave node 1 and write this value to the LEDs of the master node.	Decrease SDO entry 2000H (LED value) by 1.
PIO1_5	Read SDO entry 2000H (LED value) of slave node 2 and write this value to the LEDs of the master node.	Increase SDO entry 2000H (LED value) by 1.

Note: If no CAN bus analyzer is connected and the master node is not connected to a terminal, the effect of the heartbeat failure is hard to see.

In case of a heartbeat failure of a slave node, the effect of resetting the communication can be seen as the LEDs reset to their default value (0x00, all LEDs off).

In case of a heartbeat failure of the master node, the effect cannot be seen without a CAN bus analyzer or serial connection.

4. CANopen in more detail

Chapter 1.2 describes some basics of CANopen, this chapter will explore it in detail. Central concept in CANopen is the Object Dictionary (OD), which will be explained first. Several protocols are supported; the type of protocol used in the frame can be determined by looking at the COB-ID (Chapter 1.2). The OD can be accessed by the SDO protocol, which is described next. Finally more info regarding the NMT protocol and the heartbeat protocol are given.

These functions are explained with the aid of screenshots of the CAN bus analyzer while sniffing the traffic generated by the CANopen demo.

4.1 Object dictionary

As explained in chapter 1.2, the OD contains all parameters describing the device and its network behavior. The OD can be seen as a big register, containing a number of items which can be referenced to by a 16 bit index and an 8 bit sub index.

The OD has a defined structure, as can be seen in Table 2.

Table 2. General CANopen Object Dictionary structure

CANopen object dictionary	
Index	**Object**
0000	*not used*
0001 – 001F	Static Data Types (standard data types, e.g., Boolean, Integer16)
0020 – 003F	Complex Data Types (predefined structures composed of standard data types, e.g., PDOCommPar. SDOParameter)
0040 – 005F	Manufacturer Specific Complex Data Types
0060 – 007F	Device Profile Specific Static Data Types
0080 – 009F	Device Profile Specific Complex Data Types
00A0 – 0FFF	*reserved*
1000 – 1FFF	Communication Profile Area (e.g., Device Type, Error Register, Number of PDOs supported)
2000 – 5FFF	Manufacturer Specific Profile Area
6000 – 9FFF	Standardized Device Profile Area (e.g. “DSP-401 Device Profile for I/O Modules” [3]: Read State 8 Input Lines, etc.)
A000 – FFFF	*reserved*

Entries in the OD can be marked as RO (ReadOnly), WO (WriteOnly) and RW (Read/Write).

4.2 The COB-ID

As explained in chapter 1.1, a CAN frame consists of 11 identifier bits and 8 data bytes. In CANopen, the 11 identifier bits are known as the CAN Object Identifiers (COB-ID). The COB-ID specifies the CANopen protocol used for the current CAN frame. Table 3 shows the predefined COB-IDs.

Table 3. CANopen predefined CAN object identifiers

COB-ID	Object	Mandatory/optional
0x000	NMT	Mandatory
0x080	SYNC	Optional
0x081 – 0x0FF	EMERGENCY	Optional
0x100	TIME STAMP	Optional
0x181 – 0x1FF	PDO1 (transmit)	Optional
0x201 – 0x27F	PDO1 (receive)	Optional
0x281 – 0x2FF	PDO2 (transmit)	Optional
0x301 – 0x37F	PDO2 (receive)	Optional
0x381 – 0x3FF	PDO3 (transmit)	Optional
0x401 – 0x47F	PDO3 (receive)	Optional
0x481 – 0x4FF	PDO4 (transmit)	Optional
0x501 – 0x57F	PDO4 (receive)	Optional
0x581 – 0x5FF	SDO (transmit / server)	Mandatory
0x601 – 0x67F	SDO (receive / client)	Mandatory
0x701 – 0x77F	NMT Error Control	Mandatory

4.3 The SDO protocol

The SDO protocol is used for storing data in- and retrieving data from the OD. Any node in the network may ask another node to read or store data by SDO. The initiator is referred to as “client” and the node to which the request has been sent to, is referred to as “server”. Fig 10 shows a graphical representation.

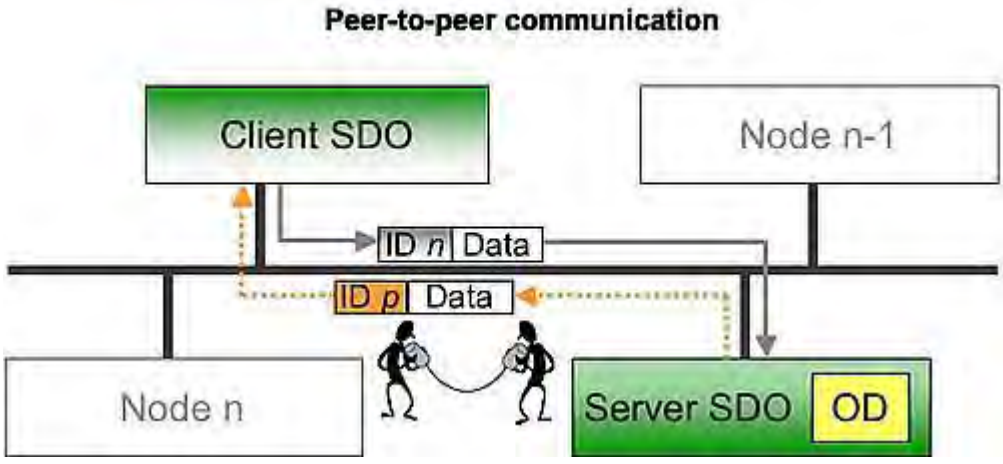

Fig 10. Object Dictionary definition of slave node

As it can be seen in Table 3, COB-IDs 0x601-0x67F are used by the client to communicate with the server and COB-IDs 0x581-0x5FF are used by the server to communicate with the client. The required COB-ID can be calculated as follows:

Client: *COB-ID = 0x600 + [Node ID of server to reach]* (broadcast not allowed)

Server: *COB-ID = 0x580 + [Node ID of client to reach]* (broadcast not allowed)

Two types of SDO access exist:

- Expedited SDO. This type of access is used to access any OD entry with size ≤ 4 bytes.
- Segmented SDO. This type of access is used to access any OD entry with size > 4 bytes, with theoretically up to an infinite number of bytes.

The SDO protocol specifies five commands:

- SDO segment download.
- Initiating download.
- Initiating upload.
- SDO segment upload.
- Abort SDO transfer.

With these five commands data can be read from- and written to the OD of an SDO server. Two types of data structures are used for these five commands:

Format 1: Used for expedited transfer, or for initiating a segmented transfer

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs	reserved(=0)	n	e	s	index	subindex	data

cs: Command Specifier. Indicates which command is used:

0: SDO segment download.

1: Initiating download.

2: Initiating upload.

3: SDO segment upload.

4: SDO abort transfer.

n: Indicates the number of data bytes which do not contain data, only if both *e* and *s* are set.

e: If set, indicates an expedited transfer (data is encapsulated in the 4 data bytes of the CAN frame). If cleared, transfer is of segmented type.

s: If set, data size is indicated in *n* (*e* is set) or data size is indicated in *data* (*e* is cleared).

index: index of OD entry.

subindex: sub index of OD entry

data: if *e* is set, *data* contains the data to transfer. If *e* is cleared and *s* is set, indicates the data size of the segmented transfer.

Format 2: Used for transferring segments of data while a segmented transfer is in progress

3 bits	1 bit	3 bits	1 bit	7 bytes
cs	t	n	l	data

cs: Command Specifier. Indicates which command is used:

0: SDO segment download.

1: Initiating download.

2: Initiating upload.

3: SDO segment upload.

4: SDO abort transfer.

t: Toggle. This bit should toggle on every frame of the segmented transfer.

n: Indicates the number of data bytes which do not contain data.

l: Last frame. Indicates for segmented transfer that this is the last frame.

data: Data bytes of segmented transfer.

4.3.1 Expedited read

To perform an expedited read of OD entry 2000h, sub index 0x00, the client issues an "initiate upload" command:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 2	reserved(=0)	n = 0	e = 0	s = 0	Index = 0x2000	Subindex = 0x00	Data = 0x00

cs = 2 indicates an upload request.

n = 0, as *s* = 0.

e = 0, not used as no data is transferred, only a request.

s = 0, no size indicated (size is not mandatory for reading).

Index = 0x2000, transfer of index 2000h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0x00, should be zero when reading.

The server should respond with the following reply, assuming the entry exists and has 1 data byte containing 0xAA:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 2	reserved(=0)	n = 3	e = 1	s = 1	Index = 0x2000	Subindex = 0x00	Data = 0xAA

cs = 2 indicates an upload.

n = 3, as only 1 data bytes is present and thus 3 data bytes are unused.

e = 1, indicating expedited transfer.

s = 1, size is indicated in *n*.

Index = 0x2000, transfer of index 2000h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0xAA, value of OD entry 0x2000, sub index 0x00 is 0xAA.

A capture of an expedited read can be found in Fig 11.

Time	Type	ID	DLC	Data
4.7833	Data	601	8	40 00 20 00 00 00 00 00
4.7843	Data	581	8	4F 00 20 00 AA 00 00 00

Fig 11. Capture from master node reading entry 2000h of slave node 1

4.3.2 Expedited write

To perform an expedited write of OD entry 2000h, sub index 0x00 with a single data byte 0x55, the client issues an “initiate download” command:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 1	reserved(=0)	n = 3	e = 1	s = 1	Index = 0x2000	Subindex = 0x00	Data = 0x55

cs = 1 indicates an “initiating download” command.

n = 3, as only 1 data bytes is written and thus 3 data bytes are unused.

e = 1, indicating an expedited transfer.

s = 1, size is indicated in *n*.

Index = 0x2000, transfer of index 2000h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0x55, byte to be written.

The server should respond with the following reply, assuming the entry exists:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 3	reserved(=0)	n = 0	e = 0	s = 0	Index = 0x2000	Subindex = 0x00	Data = 0x00

cs = 3 indicates an confirmation of the download.

n = 0, as *s* is 0.

e = 0, as no data is transferred.

s = 0, size is not indicated, as no data is transferred.

Index = 0x2000, transfer of index 2000h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0x00, as no data is transferred.

A capture of an expedited write can be found in Fig 12.

Time	Type	ID	DLC	Data
10.3294	Data	601	8	2F 00 20 00 55 00 00 00
10.3304	Data	581	8	60 00 20 00 00 00 00 00

Fig 12. Capture from master node writing single data byte “0x55” to OD entry 2000h of slave node 1

4.3.3 Segmented read

1. To perform a segmented read of OD entry 2200h, sub index 0x00, the client first issues an “initiate upload” command:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 2	reserved(=0)	n = 0	e = 0	s = 0	Index = 0x2200	Subindex = 0x00	Data = 0x00

cs = 2 indicates an upload request.

n = 0, as *s* = 0.

e = 0, not used as no data is transferred, only a request.

s = 0, no size indicated (size is not mandatory for reading).

Index = 0x2200, wants to read from index 2200h.

Subindex = 0x00, wants to read from sub index 0x00.

Data = 0x00, should be zero when reading.

2. The server should respond with the following reply, assuming the entry exists:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 2	reserved(=0)	n = 0	e = 0	s = 0	Index = 0x2200	Subindex = 0x00	Data = 0x00

cs = 2 indicates an upload.

n = 0, as *s* is 0.

e = 0, indicating a segmented transfer.

s = 0, size is not indicated.

Index = 0x2200, transfer of index 2200h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0x00, as no data in transferred in this frame.

3. The client needs to send an acknowledge to the server:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 3	t = 0	n = 0	l = 0	data = 0x00

cs = 3, indicating an acknowledge.

t = 0, toggle bit starts at 0.

n = 0, not used with acknowledge.

l = 0, not used with acknowledge.

data = 0x00, not used with acknowledge.

4. The server responds with the first set of data:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 0	t = 0	n = 0	l = 0	data = ??

cs = 0, indicating SDO segment download.

t = 0, toggle bit starts at 0.

n = 0, all 7 data bytes contain data.

l = 0, this frame is not the last frame for the segmented transfer.

data is loaded with the data present at index 2200h, sub index 0x00.

5. Steps 3 and 4 are repeated (with the *t* bit been toggled every frame) until the server is ready to send the last frame.
6. When ≤7 data bytes are left for transfer, the server sends the last frame:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 0	t = ??	n = ??	l = 1	data = ??

cs = 0, indicating SDO segment download.

t needs to be set to the inversed value of the previous frame.

n = 7 - [number of data bytes present in *data*].

l = 1, this frame is the last frame for the segmented transfer.

data is loaded with the data present at index 2200h, sub index 0x00.

A capture of a segmented read can be found in Fig 13.

Time	Type	ID	DLC	Data
11.0815	Data	601	8	40 00 22 00 00 00 00 00
11.0825	Data	581	8	40 00 22 00 00 00 00 00
11.0835	Data	601	8	60 00 00 00 00 00 00 00
11.0844	Data	581	8	00 42 6F 6F 74 2D 75 70
11.0854	Data	601	8	70 00 00 00 00 00 00 00
11.0864	Data	581	8	10 20 76 61 6C 75 65 20
11.0874	Data	601	8	60 00 00 00 00 00 00 00
11.0883	Data	581	8	00 6F 66 20 53 44 4F 20
11.0894	Data	601	8	70 00 00 00 00 00 00 00
11.0903	Data	581	8	10 32 32 30 30 68 00 00
11.0914	Data	601	8	60 00 00 00 00 00 00 00
11.0924	Data	581	8	00 00 00 00 00 00 00 00
11.0934	Data	601	8	70 00 00 00 00 00 00 00
11.0944	Data	581	8	10 00 00 00 00 00 00 00
11.0954	Data	601	8	60 00 00 00 00 00 00 00
11.0964	Data	581	8	00 00 00 00 00 00 00 00
11.0975	Data	601	8	70 00 00 00 00 00 00 00
...				
11.1556	Data	581	8	10 00 00 00 00 00 00 00
11.1566	Data	601	8	60 00 00 00 00 00 00 00
11.1576	Data	581	8	09 00 00 00 00 00 00 00

Fig 13. Capture from master node reading OD entry 2200h of slave node 1

4.3.4 Segmented write

1. To perform a segmented write of OD entry 2200h, sub index 0x00, the client first issues an “initiate download” command:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 1	reserved(=0)	n = 0	e = 0	s = 1	Index = 0x2200	Subindex = 0x00	Data = length

cs = 1 indicates an download request.

n = 0, as *e* = 0.

e = 0, not used as no data is transferred, only a request.

s = 1, size indicated in *data* (not mandatory to indicate length).

Index = 0x2200, wants to write to index 2200h.

Subindex = 0x00, wants to write to sub index 0x00.

data = number of bytes to write (not mandatory to indicate length).

2. The server should respond with the following reply, assuming the entry exists:

3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
cs = 3	reserved(=0)	n = 0	e = 0	s = 0	Index = 0x2200	Subindex = 0x00	Data = 0x00

cs = 3 indicates an acknowledge.

n = 0, as *s* is 0.

e = 0, indicating a segmented transfer.

s = 0, size is not indicated.

Index = 0x2200, transfer of index 2200h.

Subindex = 0x00, transfer of sub index 0x00.

Data = 0x00, as no data in transferred in this frame.

3. The client may now send the data to be written:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 0	t = 0	n = 0	l = 0	data = ??

cs = 0, indicating SDO segment download.

t = 0, toggle bit starts at 0.

n = 0, all 7 data bytes contain data.

l = 0, this frame is not the last frame for the segmented transfer.

data is loaded with the data to be written.

4. The server should respond with an acknowledge:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 1	t = 0	n = 0	l = 0	data = ??

cs = 1, indicating an acknowledge.

t = 0, toggle bit starts at 0.

n = 0, not used for acknowledge.

l = 0, this not used for acknowledge.

data is don't care when for acknowledge.

5. Steps 3 and 4 are repeated (with the *t* bit been toggled every frame) until the client is ready to send the last frame.
6. When ≤7 data bytes are left for transfer, the client sends the last frame:

3 bits	1 bit	3 bits	1 bit	7 bytes
cs = 0	t = ??	n = ??	l = 1	data = ??

cs = 0, indicating SDO segment download.

t needs to be set to the inversed value of the previous frame.

n = 7 - [number of data bytes present in *data*].

l = 1, this frame is the last frame for the segmented transfer.

data is loaded with the data to be written.

A capture of a segmented write can be found in Fig 14.

Time	Type	ID	DLC	Data
44.4485	Data	601	8	21 00 22 00 2C 00 00 00
44.4495	Data	581	8	60 00 22 00 00 00 00 00
44.4504	Data	601	8	00 54 68 69 73 20 69 73
44.4515	Data	581	8	20 00 22 00 00 00 00 00
44.4524	Data	601	8	10 20 61 20 6D 65 73 73
44.4534	Data	581	8	30 00 22 00 00 00 00 00
44.4544	Data	601	8	00 61 67 65 20 65 6E 74
44.4553	Data	581	8	20 00 22 00 00 00 00 00
44.4563	Data	601	8	10 65 72 65 64 20 66 72
44.4573	Data	581	8	30 00 22 00 00 00 00 00
44.4582	Data	601	8	00 6F 6D 20 74 68 65 20
44.4593	Data	581	8	20 00 22 00 00 00 00 00
44.4602	Data	601	8	10 74 65 72 6D 69 6E 61
44.4612	Data	581	8	30 00 22 00 00 00 00 00
44.4622	Data	601	8	0B 6C 00 00 00 00 00 00
44.4632	Data	581	8	20 00 22 00 00 00 00 00

Fig 14. Capture from master node writing to OD entry 2200h of slave node 1

4.4 The NMT protocol

The NMT (Network Management) protocol is used for changing the NMT state of the CANopen nodes. Any node in the network may send a NMT command to another node, or all nodes by broadcasting, in the network. The node which sends the NMT command is referred to as “Master”, while the node(s) addressed is/are referred to as “Slave(s)”. Fig 15 shows a graphical representation.

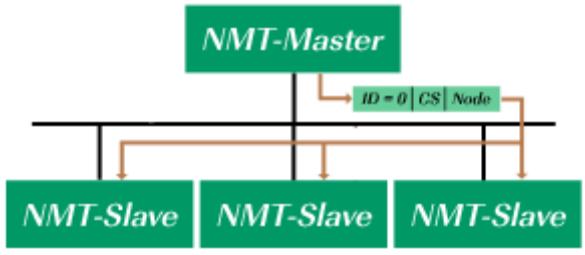

Fig 15. NMT master / slave structure

4.4.1 NMT states

Each NMT slave has a state machine, consisting of four states. These four states are shown in Fig 16.

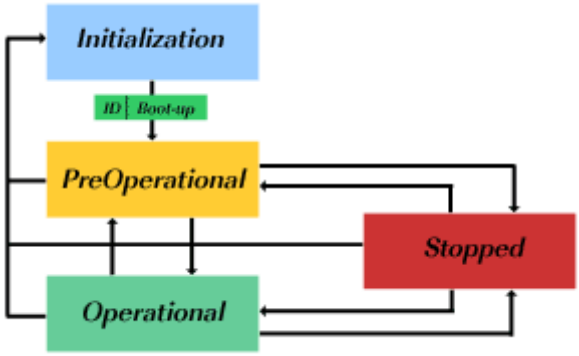

Fig 16. NMT state machine of an NMT slave

4.4.1.1 Initializing

After Power-On all NMT slaves change their state to Initializing. After internal initialization of the NMT slave, the NMT slave changes its state to Pre-Operational. Besides this state change, the NMT protocol is the only way to change the present state of an NMT slave. When changing to Pre-Operational it should send a bootup message.

The bootup message is a frame with COB-ID 0x700 + [node-id of NMT slave]. There is just one data byte encapsulated in the bootup frame, with a fixed value of 0x00.

4.4.1.2 PreOperational

In the pre-operational state SDO communication is allowed, while PDO communication is inhibited.

4.4.1.3 Operational

In the operational state both SDO and PDO communication are allowed.

4.4.1.4 Stopped

In stopped mode both SDO and PDO communication are inhibited.

4.4.2 NMT command

As stated before, an NMT master may change the state of an NMT slave by sending an NMT command. COB-ID is used for sending the command.

The NMT command has two data bytes. The first data byte is the Command Specifier (CS) and the second byte is the node-ID to specify which node is addressed.

Node-ID 0x00 is also allowed, which is the broadcast ID. On receiving an NMT command with Node-ID == 0x00, all NMT slaves should execute the NMT command.

CS can have a few predefined values:

1: Start remote node.

2: Stop remote node.

128: Enter Pre-Operational.

129: Reset node.

130: Reset Communication.

The two reset commands allow a full reset (129, reset node) or to only reset the communication parameters (130, reset communication).

4.5 The heartbeat protocol

The heartbeat protocol is part of the error control protocol and is used to check that a device is still working properly. The heartbeat protocol defines one or more nodes in the network producing a heartbeat, referred to as “heartbeat producers”. It also defines one or more nodes watching for heartbeats, referred to as “heartbeat consumers”. A node may be both producer and consumer. Fig 17 shows a graphical representation.

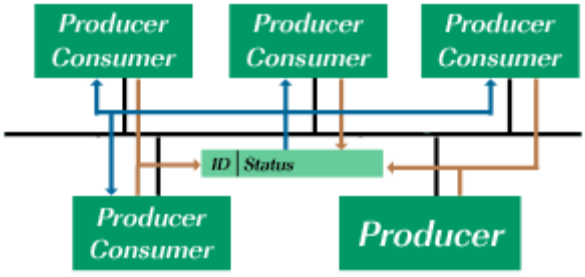

Fig 17. Heartbeat producers and consumers

A heartbeat producer sends a heartbeat frame on a regular interval (0 to 65535 ms). A heartbeat consumer listens to one or more heartbeats, and has a consumer time interval defined for every node. If the consumer did not receive a heartbeat from a node within this consumer time interval, it knows that that node is not working properly and will try to reset it by the NMT protocol.

4.5.1 The heartbeat frame

A heartbeat frame consists of a COB-ID = 0x700 + [node-id of producer] and a single data byte. The data byte represents the NMT status:

0: Boot-up (Chapter 4.4).

4: Stopped.

5: Operational.

127: Pre-Operational.

4.5.2 Associated OD entries

The heartbeat protocol is controlled by two indexes in the OD:

- 1016h. Heartbeat consumer.
- 1017h. Heartbeat producer.

4.5.2.1 Index 1016 h (Heartbeat Consumer)

OD entry 1016h is an array type. Sub index 0x00 is an entry with 1 byte in size, is ReadOnly and should be set to a value which is the maximum number of nodes which it is allowed to watch.

The other sub indexes are 4 bytes in size, are ReadWrite and are encoded as follows:

bits	31…24	32…16	15…0
Value	0	Node-ID	Heartbeat time
Encoding	-	Unsigned8	Unsigned16

Node-ID is the node-ID of the node to watch.

Heartbeat time is the heartbeat consumer interval, in milliseconds.

4.5.2.2 Index 1017h (Heartbeat Producer)

OD entry 1017h only has a valid entry at sub index 0x00. This entry is ReadWrite and has a size of 2 bytes. The value specified in these 2 bytes (encoded as Unsigned16) is the heartbeat producer interval in milliseconds.

4.5.3 Heartbeat example

The following example will demonstrate the heartbeat protocol. Please see the logging of the CAN bus while running parts of the demo in Fig 18.

1. Slave node 1 is configured by executing terminal command 1). The heartbeat producer time is set to 4000ms and the heartbeat consumer time is set to 4500ms.
2. Both the master node and the slave node are producing a heartbeat. Their status is Pre-Operational (0x7F).
3. By pressing button ISP of the slave node, a heartbeat failure of the slave node is simulated.
4. After 4500ms the master node detects the missing heartbeat and sends NMT command 0x82 (130, Reset Communication) to the slave node.
5. In reaction to NMT command 130, the slave resets its communication parameter. After completing the Initialization state, it enters the Pre-Operational state again and sends the boot-up message.

6. Master node sees the boot-up message and has remembered that there was a heartbeat failure at this node. So, it re-configures the slave node with a value for the heartbeat producer and heartbeat consumer OD entries.
7. Both the master and the slave produce a heartbeat again.

Time	Type	ID	DLC	Data	
2.2381	Data	601	8	2B 17 10 00 A0 0F 00 00	1
2.2391	Data	581	8	60 17 10 00 00 00 00 00	
2.2400	Data	601	8	23 16 10 01 94 11 7D 00	
2.2410	Data	581	8	60 16 10 01 00 00 00 00	
2.4830	Data	77D	1	7F	2
6.2380	Data	701	1	7F	
6.4829	Data	77D	1	7F	
10.2381	Data	701	1	7F	
10.4828	Data	77D	1	7F	
14.2380	Data	701	1	7F	
14.4828	Data	77D	1	7F	
18.2381	Data	701	1	7F	
18.4829	Data	77D	1	7F	
22.2381	Data	701	1	7F	
22.4829	Data	77D	1	7F	
26.4829	Data	77D	1	7F	3
26.7380	Data	000	2	82 01	4
26.7384	Data	701	1	00	5
26.7500	Data	601	8	2B 17 10 00 A0 0F 00 00	6
26.7510	Data	581	8	60 17 10 00 00 00 00 00	
26.7534	Data	601	8	23 16 10 01 94 11 7D 00	
26.7544	Data	581	8	60 16 10 01 00 00 00 00	
30.4929	Data	77D	1	7F	7
30.7502	Data	701	1	7F	
34.4929	Data	77D	1	7F	
34.7502	Data	701	1	7F	

Fig 18. Logging of the CAN bus during some actions regarding the heartbeat protocol.

5. Software

5.1 Software organization

Three IDE's are supported:

- Code Red LPCXpresso
- Keil µVision
- IAR EWARM

The source code and project files are structured as in Table 4:

Table 4. File structure used for the CANopen application

Directory/file	Description
Projects/LPCXpresso/Master.zip	Project file of master node for LPCXpresso
Projects/LPCXpresso/Slave.zip	Project file of slave node for LPCXpresso
Projects/Keil/Master/	Project directory of master node for Keil
Projects/Keil/Slave/	Project directory of slave node for Keil
Projects/IAR/Master/	Project directory of master node for IAR
Projects/IAR/Slave/	Project directory of slave node for IAR
Software/CMSIS	CMSIS source files
Software/Drivers/src/CANopen/canopen_driver.c	CANopen driver source
Software/Drivers/src/UART/uart.c	UART driver source
Software/Drivers/src/Retarget/	Source directory for retargeting printf() and scanf()
Software/Drivers/inc/CANopen/canopen_driver.h	CANopen driver header
Software/Drivers/inc/CANopen/rom_drivers.h	On-chip drivers header
Software/Drivers/inc/CANopen/rom_driver_CAN.h	On-chip CAN driver header
Software/Drivers/inc/UART/uart.h	UART driver header
Software/Master/src/main.c	Application source for master node
Software/Master/src/CAN_Node_Def.c	Node definition source file for master node
Software/Master/inc/CAN_Node_Def.h	Node definition header for master node
Software/Slave/src/main.c	Application source for slave node
Software/Slave/src/CAN_Node_Def.c	Node definition source file for slave node
Software/Slave/inc/CAN_Node_Def.h	Node definition header for slave node

5.2 Application

For both the master and the slave node most of the work is done by the CANopen driver (Chapter 5.3). The CANopen device definition (Chapter 5.4) also has an important role in the software. This chapter explains on a higher level how the software behaves.

5.2.1 Master

Fig 19 shows the flowchart of the master node.

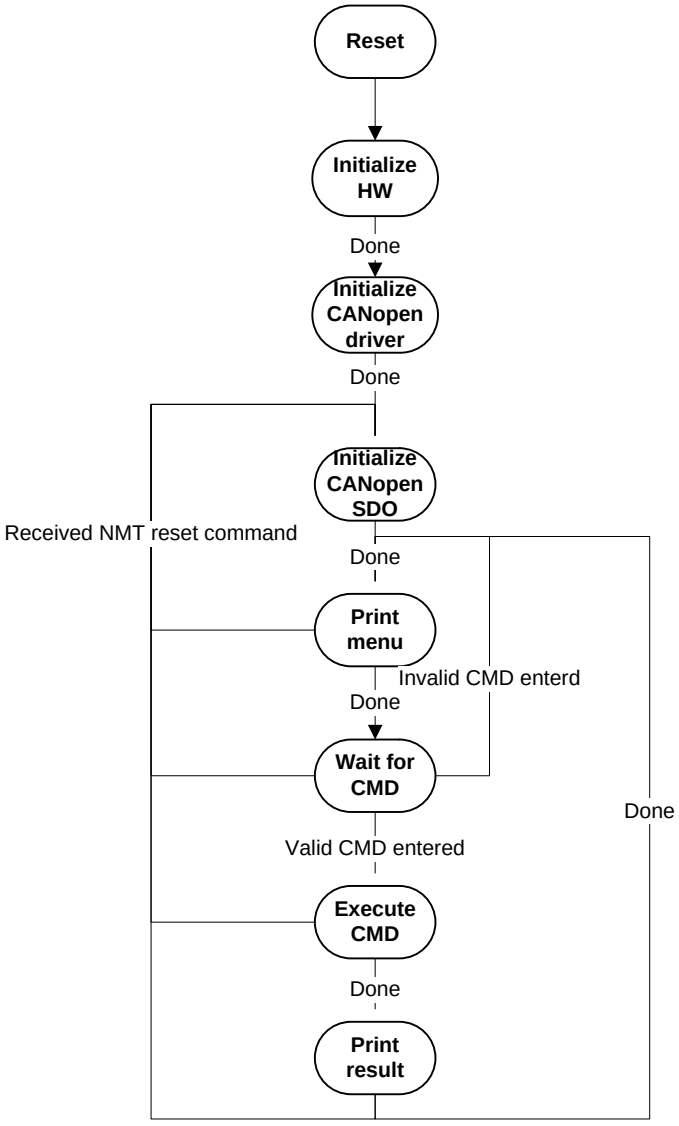

Fig 19. Flowchart of the master node

After reset, the hardware is initialized (e.g. the LPC11Cxx and its peripherals, LEDs, UART, etc.).

After the hardware initialization is done, the CANopen driver is initialized by calling the CANopenInit functions. This function makes the CANopen driver ready for use, and also calls the callback functions CANopen_Init_SDO, thereby initializing the CANopen SDO parameters. This also enables the production of the heartbeat every 4000 ms. The CANopen_Init_SDO function can be called anytime upon receiving the NMT reset command.

After the CANopen driver is fully initialized, the menu with available commands is printed. Next, a loop is entered which waits for a command. This command can come from the RS232 connection or from the buttons on the MCB11C14 board. Upon entering a valid command it will be executed (e.g. read SDO of slave and print it to the terminal). After

execution, the result is printed to the terminal, the menu is displayed and another command can be given to the master node.

5.2.2 Slave

Fig 20 shows the flowchart of the slave node.

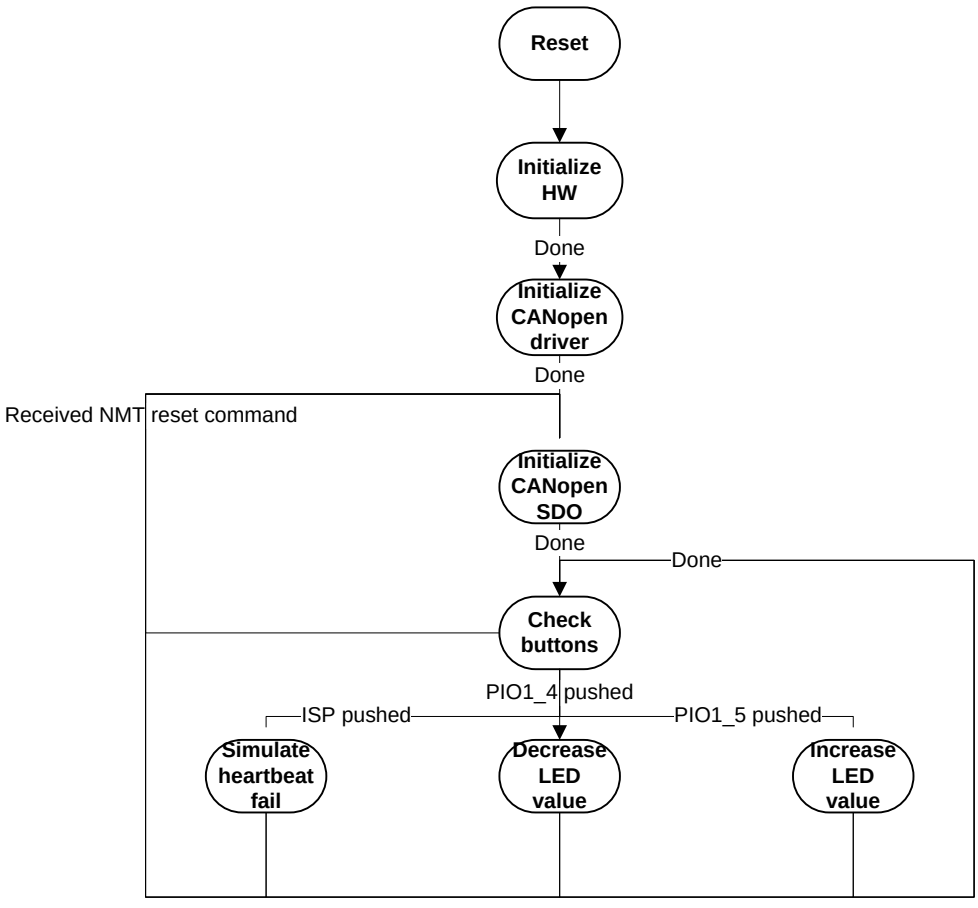

Fig 20. Flowchart of the slave node

After reset, the hardware is initialized (e.g. the LPC11Cxx and its peripherals, LEDs, UART, etc.).

After the hardware initialization is done, the CANopen driver is initialized by calling the CANopenInit functions. This function makes the CANopen driver ready for use, and also calls the callback functions CANopen_Init_SDO, thereby initializing the CANopen SDO parameters. This also enables the production of the heartbeat every 4000 ms. The CANopen_Init_SDO function can be called anytime upon receiving the NMT reset command.

After initializing the CANopen driver an endless loop is entered. In this endless loop, the software keeps checking the state of the buttons. If any of the three buttons is pressed, the appropriate action will be taken. All other functions (e.g. responding to CANopen messages and handling timeouts) are performed by the timer interrupt and the CAN interrupt.

5.3 CANopen driver

To use the CANopen driver, the file CANopen.h should be included in the project.

A number of functions are present in the API for reading or writing to other nodes. In order for the CANopen driver to work, one should have the following prerequisites in its applications:

- When the application is initializing, function CANopenInit must be called to initialize the CANopen driver and the CAN peripheral.
- Functions CANopen_1ms_tick must be called every 1ms.
- CANopen_Init_SDO callback functions must be present in the application. This functions is called when initializing the CANopen driver and on reset of the node (e.g. by NMT command).
- CANopen_NMT_Reset_Node_Received callback functions must be present in the application. This function is called on receiving the NMT command “Reset Node”.
- CANopen_NMT_Reset_Comm_Received callback functions must be present in the application. This function is called on receiving the NMT command “Reset Communication”.
- CANopen_Heartbeat_Consumer_Failed callback function must be present in the application. This functions is called when one of the node listed in the WatchList does not transmit its heartbeat on time.
- CANopen_NMT_Consumer_Bootup_Received callback functions must be present in the applications.

  Please refer to the application source code (“main.c”) of the master node and the description in the “canopen_driver.c” file to see what functions are available and how to use them.

  More information on the on-chip CAN/CANopen drivers can be found in chapter 17 of the LPC11Cxx User Manual.

5.4 CANopen device definition

The device definition defines what the OD looks like. Two files are required for a complete definition:

- CAN_Node_Def.c
- CAN_Node_Def.h

These two files define multiply of things regarding the behavior of the CANopen node, in particular:

- Its Node ID.
- Its Object Dictionary (OD).
- Variables coupled to the OD.
- SDO Client timeout value.
- Watchlist, used for initial guarding of nodes.

Fig 21 and Fig 22 show the node definition used for the slave node.

```
#ifndef _CAN_NODE_DEF_H
#define _CAN_NODE_DEF_H

#include <stdint.h>
#include "canopen_driver.h"

#define CAN_MASTER_NODE         0x7D        /* 125 */
#define CAN_SLAVE1_NODE         0x01        /* 1 */
#define CAN_SLAVE2_NODE         0x02        /* 2 */

#define CAN_NODE_ID             CAN_SLAVE1_NODE     /* make sure to change this when using 2 slaves */
#define CANOPEN_TIMEOUT_VAL     100                 /* in ms */

/* Application variables used in variable OD */
extern uint8_t  error_register;                         /* CANopen error register */
extern uint8_t  LEDArray;                               /* LEDs of MCB11C14 board */
extern uint32_t CANopen_Heartbeat_Producer_Value;       /* heartbeat producer value */
extern volatile SDOS_Buffer_t SDOS_2200;                /* buffer structure associated with segmented entry 2200h */
extern volatile uint8_t SDOS_2200_Data[255];            /* buffer associated with segmented entry 2200h */

/* Watchlist */
extern WatchNode_t WatchList[];                         /* for watching nodes */
extern uint8_t WatchListLength;                         /* number of nodes in watchlist must be known */

/* OD */
extern CAN_ODCONSTENTRY myConstOD[];                    /* constant OD entries */
extern uint32_t NumberOfmyConstODEntries;               /* required so that the number of entries in the constant OD is known */
extern CAN_ODENTRY myOD[];                              /* non-constant OD entries */
extern uint32_t NumberOfmyODEntries;                    /* required so that the number of entries in the non-constant OD is known */

#endif /* _CAN_NODE_DEF_H */
```

Fig 21. Node definition header file for slave node

```
#include "CAN_Node_Def.h"
#include <stdint.h>

/* Application variables used in variable OD */
uint8_t  error_register;
uint8_t  LEDArray;
volatile SDOS_Buffer_t SDOS_2200 =
{
    (uint8_t*)SDOS_2200_Data,
    sizeof(SDOS_2200_Data),
};
volatile uint8_t SDOS_2200_Data[255] = "Boot-up value of SDO 2200h";
uint32_t CANopen_Heartbeat_Producer_Value;
WatchNode_t WatchList[1];
uint8_t WatchListLength = sizeof(WatchList)/sizeof(WatchList[0]);



/*  CANopen read-only (constant) Object Dictionary (OD) entries
    Used with Expedited SDO only. Lengths = 1/2/4 bytes */
CAN_ODCONSTENTRY myConstOD [] =
{
    /* index, subindex, length, value */
    { 0x1000, 0x00,     4,      0x00000000UL },
    { 0x1018, 0x00,     1,      0x00000001UL },     /* only vendor ID is specified */
    { 0x1018, 0x01,     4,      0x000002DCUL },     /* NXP vendor ID for CANopen */
};
uint32_t NumberOfmyConstODEntries = sizeof(myConstOD)/sizeof(myConstOD[0]);

/*  CANopen list of variable Object Dictionary (OD) entries
    Expedited SDO with length=1/2/4 bytes and segmented SDO */
CAN_ODENTRY myOD [] =
{
    /* index, subindex, access_type | length,   value_pointer */
    { 0x1001, 0x00,     OD_EXP_RO | 1,          (uint8_t *)&error_register },
    { 0x1016, 0x00,     OD_EXP_RO | 1,          (uint8_t *)&WatchListLength},
    { 0x1016, 0x01,     OD_EXP_RW | 4,          (uint8_t *)&WatchList[0].value},
    { 0x1017, 0x00,     OD_EXP_RW | 2,          (uint8_t *)&CANopen_Heartbeat_Producer_Value},
    { 0x2000, 0x00,     OD_EXP_RW | 1,          (uint8_t *)&LEDArray},
    { 0x2200, 0x00,     OD_SEG_RW,              (uint8_t *)&SDOS_2200},
};
uint32_t NumberOfmyODEntries = sizeof(myOD)/sizeof(myOD[0]);
```

Fig 22. Node definition source file for slave node

6. References

[1] CANopen Application Layer and Communication Profile", CiA Draft Standard 301, Version 4.02

[2] LPC11C12/C14 Data sheet

[3] UM10398 – LPC111x/LPC11Cxx user manual

7. Legal information

7.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

7.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

7.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

8. Contents

1. Introduction 3
1.1 CAN 3
1.2 CANopen 3
1.3 LPC11Cxx 5
2. CANopen demo introduction 6
3. Running the demo 8
3.1 Introduction 8
3.2 Using the RS232 serial connection 8
3.3 Using the onboard buttons 12
4. CANopen in more detail 13
4.1 Object dictionary 13
4.2 The COB-ID 14
4.3 The SDO protocol 15
4.3.1 Expedited read 17
4.3.2 Expedited write 18
4.3.3 Segmented read 19
4.3.4 Segmented write 21
4.4 The NMT protocol 23
4.4.1 NMT states 23
4.4.1.1 Initializing 23
4.4.1.2 PreOperational 23
4.4.1.3 Operational 24
4.4.1.4 Stopped 24
4.4.2 NMT command 24
4.5 The heartbeat protocol 24
4.5.1 The heartbeat frame 25
4.5.2 Associated OD entries 25
4.5.2.1 Index 1016 h (Heartbeat Consumer) 25
4.5.2.2 Index 1017h (Heartbeat Producer) 25
4.5.3 Heartbeat example 25
5. Software 27
5.1 Software organization 27
5.2 Application 27
5.2.1 Master 27
5.2.2 Slave 29
5.3 CANopen driver 30
5.4 CANopen device definition 30
6. References 32
7. Legal information 33
7.1 Definitions 33
7.2 Disclaimers 33
7.3 Trademarks 33
8. Contents 34

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 12 July 2012
Document identifier: AN11238