ssCANopen User's Manual

Created by the CANopen Experts!

Version 1.4 - February 10th, 2014

© Copyright 2014 - Simma Software, Inc.

Simma Software

ssCANopen Protocol Stack License

READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE OPENING THE PACKAGE CONTAINING THE PROGRAM DISTRIBUTION MEDIA (DISKETTES, CD, ELECTRONIC MAIL), THE COMPUTER SOFTWARE THEREIN, AND THE ACCOMPANYING USER DOCUMENTATION. THIS SOURCE CODE IS COPYRIGHTED AND LICENSED (NOT SOLD). BY OPENING THE PACKAGE CONTAINING THE SOURCE CODE, YOU ARE ACCEPTING AND AGREEING TO THE TERMS OF THIS LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THE TERMS OF THIS LICENSE AGREEMENT, YOU SHOULD PROMPTLY RETURN THE PACKAGE IN UNOPENED FORM, AND YOU WILL RECEIVE A REFUND OF YOUR MONEY. THIS LICENSE AGREEMENT REPRESENTS THE ENTIRE AGREEMENT CONCERNING THE CANOPEN PROTOCOL STACK BETWEEN YOU AND SIMMA SOFTWARE, INC. (REFERRED TO AS "LICENSOR"), AND IT SUPERSEDES ANY PRIOR PROPOSAL, REPRESENTATION, OR UNDERSTANDING BETWEEN THE PARTIES.

1. Corporate License Grant. Simma Software hereby grants to the purchaser (herein referred to as the “Client”), a royalty free, non-exclusive license to use the CANopen protocol stack source code (collectively referred to as the "Software”) as part of Client’s product. Except as provided above, Client agrees to not assign, sublicense, transfer, pledge, lease, rent, or share the Software Code under this License Agreement.

2. Simma Software's Rights. Client acknowledges and agrees that the Software and the documentation are proprietary products of Simma Software and are protected under U.S. copyright law. Client further acknowledges and agrees that all right, title, and interest in and to the Software, including associated intellectual property rights, are and shall remain with Simma Software. This License Agreement does not convey to Client an interest in or to the Software, but only a limited right of use revocable in accordance with the terms of this License Agreement.

3. License Fees. The Client in consideration of the licenses granted under this License Agreement will pay a one-time license fee.

4. Term. This License Agreement shall continue until terminated by either party. Client may terminate this License Agreement at any time. Simma Software may terminate this License Agreement only in the event of a material breach by Client of any term hereof, provided that such shall take effect 60 days after receipt of a written notice from Simma Software of such termination and further provided that such written notice allows 60 days for Client to cure such breach and thereby avoid termination. Upon termination of this License Agreement, all rights granted to Client will terminate and revert to Simma Software. Promptly upon termination of this Agreement for any reason or upon discontinuance or abandonment of Client’s possession or use of the Software, Client must return or destroy, as requested by Simma Software, all copies of the Software in Client’s possession, and all other materials pertaining to the Software (including all copies thereof). Client agrees to certify compliance with such restriction upon Simma Software’s request.

5. Limited Warranty. Simma Software warrants, for Client’s benefit alone, for a period of one year (called the “Warranty Period”) from the date of delivery of the software, that during this period the Software shall operate substantially in accordance with the functionality described in the User's Manual. If during the Warranty Period, a defect in the Software appears, Simma Software will make all reasonable efforts to cure the defect, at no cost to the Client. Client agrees that the foregoing constitutes Client’s sole and exclusive remedy for breach by Simma Software of any warranties made under this Agreement. Simma Software is not responsible for obsolescence of the Software that may result from changes in Client’s requirements. The foregoing warranty shall apply only to the most current version of the Software issued from time to time by Simma Software. Simma Software assumes no responsibility for the use of superseded, outdated, or uncorrected versions of the licensed software. EXCEPT FOR THE WARRANTIES SET FORTH ABOVE, THE SOFTWARE, AND THE SOFTWARE CONTAINED THEREIN, ARE LICENSED "AS IS," AND SIMMA SOFTWARE DISCLAIMS ANY AND ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

6. Limitation of Liability. Simma Software's cumulative liability to Client or any other party for any loss or damages resulting from any claims, demands, or actions arising out of or relating to this License Agreement shall not exceed the license fee paid to Simma Software for the use of the Software. In no event shall Simma Software be liable for any indirect, incidental, consequential, special, or exemplary damages or lost profits, even if Simma Software has been advised of the possibility of such damages. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO CLIENT.

7. Governing Law. This License Agreement shall be construed and governed in accordance with the laws of the State of Indiana.

8. Severability. Should any court of competent jurisdiction declare any term of this License Agreement void or unenforceable, such declaration shall have no effect on the remaining terms hereof.

9. No Waiver. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.

TABLE OF CONTENTS

Table of Contents

SSCANOPEN USER'S MANUAL **1**

SSCANOPEN PROTOCOL STACK LICENSE 2

CHAPTER 1 INTRODUCTION **5**

CHAPTER 2 INTEGRATION OF SSCANOPEN **6**

CHAPTER 3 CAN HARDWARE ABSTRACTION LAYER **8**

3.1 DATA TYPE DEFINITIONS **8**
CAN_T 8
3.2 FUNCTION APIS **9**
CAN_INIT 9
CAN_RX 10
CAN_TX 11

CHAPTER 4 SSCANOPEN APPLICATION PROGRAM INTERFACE **12**

4.1 DATA TYPE DEFINITIONS **12**
RPDO_T 12
TPDO_T 14
INDEX_T 15
SUBINDEX_T 16
COD_T 17
COD_SUBINDEX_T 18
COB_T 19
CANOPEN_TIMESTAMP_T 20
4.1(M) DATA TYPE DEFINITIONS (MASTER ONLY) **21**
NSD_T 21
4.2 API FUNCTIONS **22**
CANOPEN_INIT 22
CANOPEN_UPDATE 23
CANOPEN_PROCESS 24
CANOPEN_TX_TPDO 25
CANOPEN_SET_ERROR 26
CANOPEN_CLEAR_ERRORS 27
CANOPEN_SET_STATE 28
CANOPEN_READ_OD 29
CANOPEN_WRITE_OD 30
CANOPEN_READ_REMOTE_OD 31

CANOPEN_WRITE_REMOTE_OD 32
CANOPEN_SYNC_ENABLE 33
CANOPEN_SYNC_DISABLE 34
4.2(M) API FUNCTIONS (MASTER ONLY) 35
CANOPEN_SET_STATE_REMOTE_NODE 35
4.2 APPLICATION CALLBACK FUNCTIONS 36
CANOPEN_APP_PROCESS 36
CANOPEN_APP_UPDATE 37
CANOPEN_APP_TIME 38
CANOPEN_APP_WILL_CHANGE_STATE 39
CANOPEN_APP_SYNC 40
CANOPEN_APP_HANDLE_SDO 41

CHAPTER 5 CONFIGURATION 42

PROJECT SETTINGS 42
DEVICE SETTINGS 43
DEVICE SETTINGS ERROR! BOOKMARK NOT DEFINED.

CHAPTER 6 EXAMPLES 47

Chapter 1
Introduction

ssCANopen is high performance CANopen protocol stack written in ANSI C. ssCANopen adheres to both the SAE CANopen specification and to the software development best practices described in the MISRA C guidelines.

ssCANopen is a modularized design with an emphasis on software readability and performance. ssCANopen is easy to understand and platform independent allowing it to be used on any CPU or DSP with or without an RTOS. ssCANopen has been shown to be up to 500% faster and 61% smaller than other commercially available CANopen protocol stacks.

ssCANopen implements the data link layer described in CIA 301. See that document for a complete understanding of the CANopen protocol and specification.

ssCANopen also implements NMT Master behavior, node boot-up process, SYNC producer, and TIME producer as described in CIA DSP-302. See that document for a complete understanding of those additional specifications.

Filenames	File Description
canopen.c	Core source file for ssCANopen. Do not modify.
canopen.h	Core header file for ssCANopen. Do not modify.
canopen_cfg.c	ssCANopen configuration file. Modification allowed.
canopen_cfg.h	ssCANopen configuration header file. Modification allowed.
canopen_app.c	End user application. Modification allowed.
canopen_app.h	End user application. Modification allowed.

Table 1-1: ssCANopen files

Chapter 2
Integration of ssCANopen

This chapter describes how to integrate ssCANopen into your application. After this is complete, you will be able to receive and transmit CANopen messages over CAN. For implementation details, please see the chapters covering the APIs for CANopen and CAN.

Integration Steps:

1. Develop or purchase a CAN device driver which adheres to the CAN API specified in Chapter 3.
2. Before using any of the CANopen module features, make sure the CAN driver has been initialized by calling can_init(). Typically it is called shortly after power-on reset and before the application is started.
3. Before using any of the ssCANopen module features, make sure the ssCANopen has been initialized by calling canopen_init(). Typically it is called after can_init() and before the application is started.
4. Call canopen_update() at a fixed periodic interval (e.g. every 10 ms). This provides the time base for the CANopen module. It is recommended that this function be called at least every 25 ms.
5. Use the configuration tool to set the key CANopen settings and export them into canopen_cfg.c and canopen_cfg.h files. These files define the CANopen device's object dictionary and key configuration values. Base master and slave profiles are provided as a starting point for creating your device profile. See the Configuration and Examples chapters for full details.
    a. Set your system's fixed periodic interval described above in step #4.
    b. Set the number of CAN networks that will be used (minimum 1).
    c. Set the device mode. Mirrored will create one CANopen device which sends and receives on all CAN networks. Independent will create an independently configured and operating CANopen device for each network.
    d. Set your CANopen node ID between 1 and 127.
    e. As needed add and/or customize entries in the object dictionary.
    f. As needed add PDO and PDO mapping entries in the object dictionary for all RPDOs and TPDOs the device will support.

   g. Export the canopen_cfg.c and canopen_cfg.h files from the configuration tool.

6. (Optional) Implement canopen_app_handle_sdo() to use SDO client functionality.
7. (Optional) Implement canopen_app_time() to support synchronizing the device to network time.

Chapter 3
CAN Hardware Abstraction Layer

The hardware abstraction layer (HAL) is a software module that provides functions for receiving and transmitting controller area network (CAN) data frames. Because CAN peripherals typically differ from one microcontroller to another, this module is responsible for encompassing all platform depended aspects of CAN communications.

The HAL contains three functions that are responsible for initializing the CAN hardware and handling buffered reception and transmission of CAN frames.

Function Prototype	Function Description
void can_init (void)	Initializes CAN hardware
uint8_t can_rx (uint8_t p, can_t *frame)	Receives CAN frame (buffered I/O)
uint8_t can_tx (uint8_t p, can_t *frame)	Transmits CAN frame (buffered I/O)

Table 3-1: HAL functions

3.1 Data Type Definitions

can_t

Data type:

can_t

Description:

can_t is a data type used to store CAN frames. It contains the CAN frame identifier, the CAN frame data, the size of data, and CAN network number.

NOTE: If the most significant bit of id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Definition:

```
typedef struct {

  uint32_t id;
  uint8_t buf[8];
  uint8_t buf_len;

} can_t;
```

3.2 Function APIs

can_init

Function Prototype:

void can_init(void);

Description:

can_init initializes the CAN peripheral for reception and transmission of CAN frames at a network speed of 250 kbps. Any external hardware that needs to be initialized can be done inside of can_init. The sample point should be as close to 0.875 as possible, but should not exceed it. See CANOPEN/11 and CANOPEN/15 for additional bit timing and sample point information.

Parameters:

void

Return Value:

void

can_rx

Function Prototype:

uint8_t can_rx (can_t *frame);

Description:

can_rx checks to see if there is a CAN data frame available in the receive buffer. If one is available, it is copied into the can_t structure which is pointed to by frame. If the most significant bit of frame->id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Parameters:

frame: Points to memory where the received CAN frame should be stored.

Return Value:

1: No CAN frame was read from the receive buffer.

0: A CAN frame was successfully read from the receive buffer.

can_tx

Function Prototype:

uint8_t can_tx (can_t *frame);

Description:

If memory is available inside of the transmit buffer, can_tx copies the memory pointed to by frame to the transmit buffer. If transmission of CAN frames is not currently in progress, then it will be initiated. If the most significant bit of frame->id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Parameters:

frame: Points to the CAN frame that should be copied to the transmit buffer.

Return Value:

1: No CAN frame was written to the transmit buffer.

0: The CAN frame was successfully written to the transmit buffer.

Chapter 4
ssCANopen Application Program Interface

This chapter describes the application program interface (API) for the CANopen module.

Function Prototypes	Function Descriptions
void canopen_init (void)	Initializes protocol stack
void canopen_update (void)	Provides periodic time base
void canopen_process (uint8_t net, canopen_t *msg)	Processes received CANopen messages
void canopen_tx_tpdo(uint8_t net, uint8_t n)	Transmits an event based TPDO
void canopen_set_error (uint8_t net, uint16_t emcy_code, uint8_t *mfg_code)	Set error state
void canopen_clear_errors (uint8_t net)	Reset error state
uint8_t canopen_set_state (uint8_t net, uint8_t canopen_nmt_state)	Set device operational state
void canopen_read_od(uint8_t net, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t *len)	Read local object dictionary entry
void canopen_write_od(uint8_t net, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t len)	Write local object dictionary entry
void canopen_read_remote_od(uint8_t net, uint8_t node_id, uint16_t index, uint8_t subindex)	Read remote object dictionary entry (SDO client)
void canopen_write_remote_od(uint8_t net, uint8_t node_id, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t len)	Write remote object dictionary entry (SDO client)
void canopen_sync_enable(uint8_t net)	Enable SYNC producer
void canopen_sync_disable(uint8_t net)	Disable SYNC producer
void canopen_set_state_remote_node(uint8_t net, uint8_t node_id, uint8_t canopen_nmt_state)	Set remote node state (Master only)

Table 4-1: API functions

4.1 Data Type Definitions

rpdo_t

Description:

rpdo_t is a data type used to store RPDO configuration information. It contains the command byte, the position in canopen_od, and the number of object dictionary entries it spans.

Definition:

```
typedef struct
{
  uint8_t com;
  uint8_t entry;
  uint8_t entries;
} rpdo_t;
```

tpdo_t

Description:

TPDO_t is a data type used to store TPDO configuration information. It contains the command byte, the position in canopen_od, and the number of object dictionary entries it spans. It also stores the timestamps for event and inhibit timers, if enabled.

Definition:

```
typedef struct
{
  uint8_t com;
  uint8_t entry;
  uint8_t entries;
  uint8_t type;
  uint16_t event_timestamp;
  uint16_t inhibit_timestamp;
} tpdo_t;
```

index_t

Description:

index_t is a data type used to store each dynamic object dictionary entry. It contains the object dictionary index (no correlation to position in array), subindex array, and the number of subindexes allocated, and up to 4 bytes of data. canopen_od is an array of this data type.

Definition:

```
typedef struct
{
  uint16_t index;
  uint8_t subindexes;
  subindex_t *subindex;
} index_t;
```

subindex_t

Description:

subindex_t is a data type used to store each dynamic object dictionary subindex entry. It contains the number of data elements and array of up to 4 bytes of data. Each index in canopen_od contains an array of this data type.

Definition:

```
typedef struct
{
  uint8_t len;
  uint8_t *data;
} subindex_t;
```

cod_t

Description:

cod_t is a data type used to store each constant object dictionary entry. It contains the index, the number of subindexes, and an array of subindex entries. canopen_od is an array of this data type.

Definition:

```
typedef struct
{
  uint16_t index;
  uint8_t num_subindexes;
  cod_subindex_t *subindexes;
} cod_t;
```

cod_subindex_t

Description:

cod_subindex_t is a data type used to store each object dictionary subindex entries. It contains a command byte and up to 32 bits of data. Each index in canopen_cod contains an array of this data type.

Definition:

```
typedef struct
{
  uint8_t cmd;
  uint8_t *data;
} cod_subindex_t;
```

cob_t

Description:

COB_t is a data type used to store a communication object (COB). It contains the function code, source address, the number of data bytes, and up to 8 bytes of data. Incoming CANopen messages are passed to the application layer using this object.

Definition:

```
typedef struct
{
  uint8_t fcode;
  uint8_t src;
  uint8_t *buf;
  uint8_t buf_len;
} cob_t;
```

canopen_timestamp_t

Description:

canopen_timestamp_t is a data type used to store and transmit the network time. It contains 28bits representing milliseconds, 4 reserved bits, and 16 bits for days since the reference time of midnight January 1, 1984. This data type is used by object dictionary index 0x1012 and the timestamp consumer/producer functions.

Definition:

```
typedef struct
{
  uint32_t milliseconds;
  uint16_t days;
} canopen_timestamp_t;
```

4.1(m) Data Type Definitions (Master only)

nsd_t

Description:

NSD_t is a data type used to store the status of nodes observed on the CANopen network. It contains the nodeid, operational state, and timestamp representing the time the last heartbeat was observed. canopen_nsd is an array of this data type.

Definition:

```
typedef struct
{
  uint8_t nodeid;
  uint8_t state;
  uint8_t flags;
  uint32_t lastseen_timestamp;
} nsd_t;
```

4.2 API Functions

canopen_init

Function Prototype:

void canopen_init (void);

Description:

Initializes the CANopen module.

Parameters:

void

Return Value:

void

canopen_update

Function Prototype:

void canopen_update (void);

Description:

Provides the periodic time base for the CANopen module. This function should be called by the application at the period defined in CANOPENCFG_TICK_PERIOD.

Parameters:

void

Return Value:

void

canopen_process

Function Prototype:

void canopen_process (uint8_t net, canopen_t *msg);

Description:

Processes received CANopen message. This function is called by the CANopen module with a complete CANopen message and is the intended location for the application layer to handle received CANopen messages. Multipacket messages are not supported in this version.

Parameters:

net: CAN network.

msg: Pointer to received CANopen message.

Return Value:

void

canopen_tx_tpdo

Function Prototype:

void canopen_tx_tpdo (uint8_t net, uint8_t n);

Description:

This function can be called from the application to send an event based TPDO. Note: TPDOs can also be transmitted from canopen_update() under the context of either inhibit time or a SYNC message. For event time, is_changed is always set to 1, but inhibit time actually checks if the PDO data has changed before transmitting.

Parameters:

net: CAN network.

n: Designates which TPDO to transmit.

is_changed: This parameter allows other parts of the code to manually flag this TPDO as changed. Under canopen_update, is_changed is set to 1 with regard to event time and set to 0 with regard to inhibit time.

Return Value:

1: Message was not buffered for transmission.

0: Message was buffered for transmission.

canopen_set_error

Function Prototype:

void canopen_set_error (uint8_t net, uint16_t emcy_code, uint8_t *mfg_code);

Description:

Call to set the error state indicating an internal error has occurred. This will add the error to the error field (object 1003h) and transmit the error on the CANopen network using the EMCY object. Supporting the EMCY object is optional however, if implemented, the device should support at minimum the 0000h (Error reset) and 1000h (Generic error) codes.

See canopen_clear_errors(…) for clearing an error state.

Parameters:

net: CAN network.

emcy_code: CANopen EMCY code best representing internal error. See canopen.h for a partial list and CIA 301 7.2.7.1 for the complete list.

mfg_code: An optional 5-byte manufacture specific error code.

Return Value:

void

canopen_clear_errors

Function Prototype:

void canopen_clear_errors (uint8_t net);

Description:

Call to clear the CANopen device's error state. This will clear the error field (object 1003h) and transmit an EMCY message with the CANOPEN_EMCY_ERROR_RESET code signaling to EMCY consumers that the error condition has been cleared.

Parameters:

net: CAN network.

Return Value:

void

canopen_set_state

Function Prototype:

uint8_t canopen_set_state (uint8_t net, uint8_t canopen_nmt_state);

Description:

Requests a change in the operational state of the CANopen device. See diagram “operational states” for permissible state changes. The state change will only occur if it is a permitted state change. Example: A device in the Initialization state cannot switch to Operational directly.

Parameters:

net: CAN network.

canopen_nmt_state: state to transition to. See CANOPEN_STATE_* in canopen.h.

Return Value:

uint8_t: success code (0=success, 1=fail)

canopen_read_od

Function Prototype:

void canopen_read_od(uint8_t net, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t *len);

Description:

Read the object dictionary location specified by index and subindex into buf.

Parameters:

net: CAN network.

index: index in the remote node's object dictionary

subindex: subindex in the remote node's object dictionary

buf: buffer to store read value

len: bytes to read

Return Value:

void

canopen_write_od

Function Prototype:

void canopen_write_od(uint8_t net, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t len);

Description:

Writes the value in buf to the object dictionary location specified by index and subindex.

Parameters:

net: CAN network.

node_id: node id of the remote node

index: index in the remote node's object dictionary to modify

subindex: subindex in the remote node's object dictionary to modify

buf: data bytes to write

len: number of data bytes to write

Return Value:

void

canopen_read_remote_od

Function Prototype:

void canopen_read_remote_od(uint8_t net, uint8_t node_id, uint16_t index, uint8_t subindex);

Description:

Read the object dictionary value at a given location in a remote node's object dictionary. Node ID, index, and subindex values indicate the node and location of the value to read.

This function requests an SDO transfer from the remote node's SDO server. If the data is less than 4 bytes an expedited transfer is used, if length exceeds 4 bytes segmented transfer is used. Block transfer is not supported. The returned data is passed to canopen_app_handle_sdo().

Parameters:

net: CAN network.

node_id: node id of the remote node

index: index in the remote node's object dictionary to read

subindex: subindex in the remote node's object dictionary to read

Return Value:

void

canopen_write_remote_od

Function Prototype:

void canopen_write_remote_od(uint8_t net, uint8_t node_id, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t len);

Description:

Changes the value at a given location in a remote object dictionary. Node ID specifies the remote node, index and subindex indicate the locate in the object dictionary.

This function initiates an SDO transfer to the remote node's SDO server to transfer the data. If the data is less than 4 bytes an expedited transfer is used. If the data exceeds 4 bytes, segmented transfer is used. Block transfer is not supported.

Parameters:

net: CAN network.

node_id: node id of the remote node

index: index in the remote node's object dictionary to modify

subindex: subindex in the remote node's object dictionary to modify

buf: data bytes to write

len: number of data bytes to write

Return Value:

void

canopen_sync_enable

Function Prototype:

void canopen_sync_enable(uint8_t net);

Description:

Enables the node to begin transmitting the SYNC message according to the configured SYNC inhibit period. Only one node on a network should be a SYNC producer.

Parameters:

net: CAN network.

Return Value:

void

canopen_sync_disable

Function Prototype:

void canopen_sync_disable(uint8_t net);

Description:

Disables SYNC message transmission.

Parameters:

net: CAN network.

Return Value:

void

4.2(m) API Functions (Master only)

canopen_set_state_remote_node

Function Prototype:

void canopen_set_state_remote_node(uint8_t net, uint8_t node_id, uint8_t canopen_nmt_state);

Description:

Requests a change in the operational state of the remote CANopen device. See diagram “operational states” for permissible state changes. The state change will only occur if it is a permitted state change. Example: A device in the Initialization state cannot switch to Operational directly.

Parameters:

net: CAN network.

node_id: node id of the remote node. Use 0 to affect all nodes.

canopen_nmt_state: state to transition to. See CANOPEN_STATE_* in canopen.h.

Return Value:

void

4.2 Application Callback Functions

canopen_app_process

Function Prototype:

void canopen_app_process (uint8_t net, cob_t *cob);

Description:

This function is called from the CANopen stack with every CANopen message while the device is operational. cob contains the function code, source address, the number of data bytes, and up to 8 bytes of data.

Use this function to process incoming PDO data and EMCY messages.

Parameters:

net: CAN network.

cob: communication object.

Return Value:

void

canopen_app_update

Function Prototype:

void canopen_app_update (void);

Description:

This function is called periodically from the CANopen stack.

Use this function to process incoming PDO data and EMCY messages.

Parameters:

void

Return Value:

void

canopen_app_time

Function Prototype:

void canopen_app_time (uint8_t net, canopen_timestamp_t *ts);

Description:

This function is called with every TIME message received by the CANopen stack. Use this function to synchronize the node's clock with the network time.

The TIME message is given a CAN-ID with high priority however there still may be some latency due to other messages being transmitted first. If additional precision is required the high-resolution timestamp object (1013h) can be mapped into a PDO.

Parameters:

net: CAN network.

ts: timestamp containing days and milliseconds since epoch (midnight Jan. 1, 1984)

Return Value:

void

canopen_app_will_change_state

Function Prototype:

void canopen_app_will_change_state (uint8_t net, uint8_t canopen_state_current, uint8_t canopen_state_final);

Description:

This function is called when the CANopen device is changing operational state.

Parameters:

net: CAN network.

canopen_state_current: current state

canopen_state_final: destination state

Return Value:

void

canopen_app_sync

Function Prototype:

void canopen_app_sync (uint8_t net, uint8_t counter);

Description:

This function is called from the CANopen stack when a SYNC message is received. Use this function to synchronize behavior across CANopen nodes. Sample data here in preparation for a synchronous PDO transmission. See CiA 7.2.2.2 for more information about synchronous PDO transmission.

The counter byte may be implemented on devices implementing CiA 301 v4.1. The counter allows for multiple virtual SYNC messages on the same CANopen network to distribute bus load or have varied SYNC behavior across nodes.

Parameters:

net: CAN network.

counter: one byte of counter data (may be implemented in some devices).

Return Value:

void

canopen_app_handle_sdo

Function Prototype:

void canopen_app_handle_sdo (uint8_t net, uint8_t nodeid, uint16_t index, uint8_t subindex, uint8_t *buf, uint8_t buf_len);

Description:

This function is called from the CANopen stack when an SDO response has been received.

Parameters:

net: CAN network.

nodeid: node ID of remote SDO server

index: index of object dictionary entry

subindex: subindex of object dictionary entry

buf: data bytes

buf_len: number of data bytes

Return Value:

void

Chapter 5
Configuration

This chapter describes all configurable items of the CANopen module. All of these configurations can be set using the configuration tool application which has been designed to make configuration of the object dictionary easier. The application will export all settings and the object dictionary to two C language files, canopen_cfg.c and canopen_cfg.h.

Requirements

Environment

The configuration tool is designed to operate on Windows 7 and emit standard C files for inclusion with the CANopen stack files. Projects can be saved from the configuration tool to a text-based format.

Some systems, including Windows XP, will require .NET 4.0 to be installed from the following link: http://www.microsoft.com/en-us/download/details.aspx?id=17851

Project Settings

Figure 5.1 – Configuring project settings.

CANopen System Tick Period Configuration

After configuring the system to call canopen_update at a fixed periodic interval the definition CANOPENCFG_TICK_PERIOD should be set to match the number of

milliseconds in the period. This provides the time base for the CANopen module. It is recommended that this function be called at least every 25 ms.

```
/* System's fixed tick period in milliseconds */
#define CANOPENCFG_TICK_PERIOD        20U
```

Export Filename

Set the path and filename for the exported CANopen settings file. Defaults are canopen_cfg.c / canopen_cfg.h.

```
Export filename: filename and path
```

Developer

The developer fields version and notes are provided solely for the developer's use and could be used to track the version of the device profile and a descriptive comment.

```
version: Text for developer's use
notes: Text comment for developer's use
```

CAN

Configure the number of CAN networks (minimum 1) then select the device mode. Mirrored will create one CANopen device which sends and receives on all CAN networks. Independent will create an independently configured and operating CANopen device for each network.

```
CAN_NETWORKS: number of CAN networks to use
DEVICE_MODE: 0=mirrored, 1=independent
```

Device Settings

CANopen node ID

The node ID identifies the CANopen device's unique address on the network. The node ID must be between 1 and 127.

```
Nodeid: number between 1 and 127
```

NMT Master

This flag enables NMT master functionality in the given node. Only one node on a CANopen network can be master. (Requires ssCANopen-master)

```
NMT_MASTER: 0=disabled or 1=enabled
```

CANopen Heartbeat

Configure the interval at which heartbeats are transmitted from this node in milliseconds. Heartbeat must be greater than 0 (Node-guarding not supported).

```
#define OD_HB_TIME     (3000U/CANOPENCFG_TICK_PERIOD)
```

SYNC Producer

This flag enables SYNC producer functionality in the given node. Only one node on a CANopen network can be a SYNC producer (not necessarily the master).

SYNC communication cycle period is the period in microseconds of the SYNC signal.

SYNC_MAX_VALUE sets the highest value for the counter (object 1019h). This should be a multiple of the highest synchronous PDO cycle to guarantee that the lowest frequency synchronous PDO will be transmitted in one counter cycle.

```
SYNC_PRODUCER:      0=disabled or 1=enabled
SYNC_CYCLE_TIME:    (3000U/CANOPENCFG_TICK_PERIOD)
SYNC_MAX_VALUE:     100
```

Object Dictionary

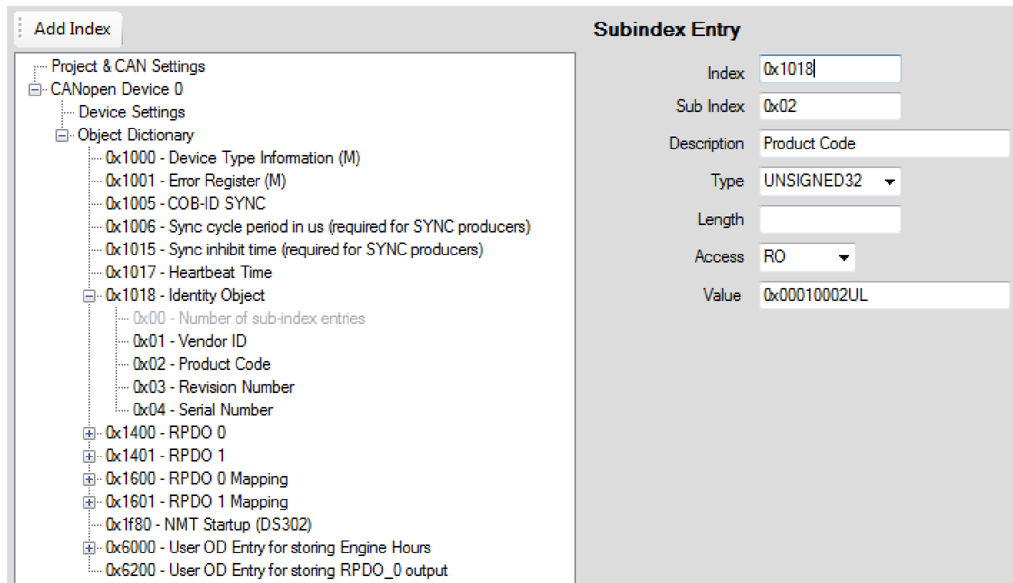

Figure 5.2 – Configuring object dictionary index 1018h subindex 0x02 (Product Code)

Object Dictionary

The object dictionary values are set automatically based on the number of object dictionary entries, TPDOs, and RPDOs that are defined using the configuration tool.

```
#define CANOPEN_CFG_NUMBER_TPDOS      0
#define CANOPEN_CFG_NUMBER_RPDOS      1
#define CANOPEN_CFG_MAX_PDO_SIZE      8
#define CANOPEN_ COD_SIZE             42
#define CANOPEN_ OD_SIZE              12
```

CANopen Proprietary Application Settings

These are miscellaneous fields representing the node id, device type, vendor id, product code, product revision and serial number. See the CANopen specification, CIA 301, for details. These are stored in objects 1001h and 1018h.

```
#define CANOPEN_NODE_ID            0x01
#define OD_DEVICE_TYPE             0x000F0191L
#define OD_VENDOR_ID               0x00455341L
#define OD_PRODUCT_CODE            0x00010002L
#define OD_REVISION                0x00010020L
#define OD_SERIAL                  0xFFFFFFFFL
```

CANopen Special Object Dictionary Entries

These definitions indicate to the program the position object dictionary entries such as heartbeat and error register. Do not modify these values.

```
#define HB                         0
#define ER                         1
```

CANopen PDO Entry Layout

These definitions define the location of PDO value in the object dictionary. These definitions are used by canopen.c. Do not modify these values.

```
#define PDO_COB_ID                 1
#define PDO_TYPE                   2
#define PDO_INHIBIT                3
#define PDO_EVENT                  4
```

Chapter 6
Examples

This chapter breaks down example(s) which provide an example of how to implement the ssCANopen protocol stack.

6. Example: Slave – Engine Hours

The slave example demonstrates a node that simulates a measured value called engine hours and reports them back to a master node via an asynchronous PDO transmission.

Engine hours are stored in the slave object dictionary at index 0x6000 subindex 0x01. The engine hour value is updated periodically in app_update().

```
uint8_t init_engine_hours[4] = {0, 0, 0, 0};
canopen_write_od( 0x6000, 0x01, init_engine_hours, 4 )
```

TPDO_1 on the slave is configured to transmit on the default address 0x182 (0x180 + 0x02).

6(m). Example: Master

The slave example demonstrates an NMT master node that is designed to receive “engine hour” transmissions from the example slave. After initializing the master resets all of the nodes on the network then after receiving the init heartbeat from the slave node in canopen_app_process(), the master commands the example slave to go pre-operational and finally operational.

RPDO_0 on the master configured to receive the message from the slave. Received values are stored at index 0x6200 subindex 0x01. Periodically in canopen_app_update() the master reads the stored value from that location and places it temporarily in a variable called hours.