LPC11xx CAN ISP and API Specification

Revision 0.12

Date updated: 1/22/2010

Contents

Chapter 1 - Introduction

1.1 About CAN ISP

The purpose of the CAN ISP is to add flash programming capability to the NXP LPC11xx family, using CAN as an alternative communication channel to the readily supported UART.

The suggested implementation is using the CANopen SDO protocol and data organization method. However, the currently existing command structure for the UART ISP commands is mapped into this method for clarity and simplicity as much as feasible.

The developed code will become part of the ROM library in the LPC11xx parts.

1.2 About CAN API

The developed code offers it’s CAN and CANopen initialization and communication features to user applications via a defined API. It includes functions such as:

- CAN set-up and initialization
- CAN send and receive messages
- CAN status
- CANopen Object Dictionary
- CANopen SDO expedited communication
- CANopen SDO segmented communication primitives
- CANopen SDO fall-back handler

Chapter 2 – CAN ISP

2.1 Execution and Operation of the CAN ISP

The CAN bootloader is activated by the ROM reset handler automatically, based on a yet to be defined hardware condition. It initializes the on-chip oscillator and the CAN controller for a CAN bitrate of 100 kbps and sets its own CANopen Node ID to a fixed value. It then waits for (CANopen SDO) commands and responds to them. These commands allow to read and write anything in a so-called Object Dictionary (OD). The OD contains entries that are addressed via a 16-bit index and 8-bit subindex. The command interface is part of this OD. It allows to perform all functions that are otherwise available via the UART ISP commands. In addition, it allows to reconfigure the clock circuitry and CAN controller for a different CAN bitrate via a low-level, register-oriented interface. Higher bitrates than 100 kbps require an external crystal. The host has to have all knowledge about the the chip's clock generation module and the CAN controller bitrate settings.

The SDO commands are received, processed and responded to "forever", until the command to jump to a certain execution address ("Go") has been received or the chip is reset.

2.2 Node ID and Multiple CAN ISP Devices on a CAN Bus

In the first version, the CAN ISP will occupy the fixed CANopen Node ID 125 (7Dh).

In the future, an assignment of unique CANopen Node IDs to multiple nodes that are all in ISP mode and connected on the same bus should be possible, using the Layer-Setting Service (LSS) FastScan protocol and the unique 128-bit serial number that is part of each LPC11xx chip.

2.3 CAN ISP SDO Communication

The CAN ISP node listens for CAN 2.0A (11-bit) messages with the identifier of 600h plus the Node ID, so 67Dh. It sends SDO responses with the identifier 580h plus Node ID, or 5FDh. The SDO communication protocols "expedited" and "segmented" are supported. This means that communication is always confirmed: Each request CAN message will be followed by a response message from the ISP node.

The SDO Block Transfer mode is not supported.

For details regarding the SDO protocol see the CiA 301 specification.

2.4 CAN ISP Object Dictionary

The complete Object Dictionary will look as follows:

Index	Subindex	Data Type	Access	Description
1000	00	UNSIGNED32	RO	**Device Type** (ASCII "LPC1")
1001	00		RO	**Error Register** (not used, 0x00)
1018	00			**Identity Object**
	01	UNSIGNED32	RO	Vendor ID (not used, 0x0000000)
	02	UNSIGNED32	RO	Part Identification Number
	03	UNSIGNED32	RO	Boot Code Version Number
1F50	00			**Program Data**
	01	DOMAIN	RW	Program Area
1F51	00			**Program Control**
	01	UNSIGNED8	RW	Program Control

5000	00	UNSIGNED16	WO	**Unlock Code**
5010	00	UNSIGNED32	RW	**Memory Read Address**
5011	00	UNSIGNED32	RW	**Memory Read Length**
5015	00	UNSIGNED32	RW	**RAM Write Address**
5020	00	UNSIGNED16	WO	**Prepare Sectors for Write**
5030	00	UNSIGNED16	WO	**Erase Sectors**
5040	00			**Blank Check Sectors**
	01	UNSIGNED16	WO	Check sectors
	02	UNSIGNED32	RO	Offset of the first non-blank location
5050	00			**Copy RAM to Flash**
	01	UNSIGNED32	RW	Flash Address (DST)
	02	UNSIGNED32	RW	RAM Address (SRC)
	03	UNSIGNED16	RW	Number of Bytes
5060	00			**Compare Memory**
	01	UNSIGNED32	RW	Address 1
	02	UNSIGNED32	RW	Address 2
	03	UNSIGNED16	RW	Number of Bytes
	04	UNSIGNED32	RO	Offset of the first mismatch
5070	00			**Execution Address**
	01	UNSIGNED32	RW	Execution Address
	02	UNSIGNED8	RO	Mode ('T' or 'A'), only 'T' supported
5100	00			**Serial Number**
	01	UNSIGNED32	RO	Serial Number 1
	02	UNSIGNED32	RO	Serial Number 2
	03	UNSIGNED32	RO	Serial Number 3
	04	UNSIGNED32	RO	Serial Number 4
5200	00			**Reconfigure LPC**
	01	UNSIGNED32	RW	SYSOSCCTRL
	02	UNSIGNED32	RW	SYSPLLCTRL
	03	UNSIGNED32	RW	SYSPLLCLKSEL
	04	UNSIGNED32	RW	MAINCLKSEL
	05	UNSIGNED32	RW	SYSAHBCLKDIV
	06	UNSIGNED32	RW	CANCLKDIV

Index	Subindex	Data Type	Access	Description
	07	UNSIGNED32	RW	CAN_BTR
	08	UNSIGNED32	RW	CPU Frequency (kHz)
	09	UNSIGNED16	RW	Activate/Status

2.5 UART ISP Command Equivalents in the CAN ISP Interface

a) Unlock U <Unlock Code>

Write <Unlock Code> to [5000h,0]. Writing an invalid unlock code will return a dedicated abort code.

b) Set Baud Rate B <Baud Rate>

No direct equivalent. However, the clock cicuitry and CAN controller bitrate can be reconfigured by writing to [5200h,1-9] and then writing anything to [5200h,10]. Reading [5200h,10] returns a bitmask where a value that has been written since the last activation sets the corresponding bit (bit 0 = subindex 1 etc.).

c) Echo A <setting>

No equivalent.

d) Write to RAM W <start address> <number of bytes>

Set RAM write address by writing to [5015h,0]. Then write the binary data to [1F50h,1]. Since this is a DOMAIN entry, the data can be continuously written. The host terminates the write. The write address in [5015h,0] auto-increments, so a write of a larger area may be done in multiple successive write cycles to [1F50h,1].

e) Read Memory R <address> <number of bytes>

Set RAM read address by writing to [5010h,0] and the read length by writing to [5011h,0]. Then read the binary data from [1F50h,1]. Since this is a DOMAIN entry, the data is continuously read. The device terminates the read when the number of bytes in the read length entry has been read. The read address in [5010h,0] auto-increments, so a read of a larger area may be done in multiple successive read cycles from [1F50h,1].

f) Prepare sector(s) for write operation P <start sector number> <end sector number>

Write a 16-bit value to [5020h,0] with the start sector number in the lower eight bits and the end sector number in the upper eight bits.

g) **Copy RAM to Flash C <flash address> <RAM address> <number of bytes>**

   Write the parameters into entry [5050h,1-3]. The write of the number of bytes into [5050h,3] starts the programming.

h) **Go G <address> <Mode>**

   Write the start address into [5070h,0]. Then trigger the "start application" command by writing the value 1h to [1F51,1].

i) **Erase sector(s) E <start sector number> <end sector number>**

   Write a 16-bit value to [5030h,0] with the start sector number in the lower eight bits and the end sector number in the upper eight bits.

j) **Blank check sector(s) I <start sector number> <end sector number>**

   Write a 16-bit value to [5040h,1] with the start sector number in the lower eight bits and the end sector number in the upper eight bits.

   If the SECTOR_NOT_BLANK abort code is returned, the entry [5040h,2] contains the offset of the first non-blank location.

k) **Read Part ID J**

   Read [1018h,2].

l) **Read Boot Code version K**

   Read [1018h,3].

m) **Read serial number N**

   Read [5100h,1-4]

n) **Compare M <address1> <address2> <number of bytes>**

   Write the parameters into entry [5060h,1-3]. The write of the number of bytes into [5060h,3] starts the comparison.

   If the COMPARE_ERROR abort code is returned, the entry [5060h,4] can be read to get the offset of the first mismatch.

2.6 CAN ISP SDO Abort Codes

The OD entries that trigger an action return an appropriate SDO abort code when the action returned an error. The abort code is 0F00’0000h plus the value of the corresponding ISP return code in the lowest byte. The list of abort codes is as follows:

UART ISP Error Code	SDO Abort Code	Value
ADDR_ERROR	SDOABORT_ADDR_ERROR	0F00 000Dh
ADDR_NOT_MAPPED	SDOABORT_ADDR_ NOT_MAPPED	0F00 000Eh
CMD_LOCKED	SDOABORT_CMD_LOCKED	0F00 000Fh
CODE_READ_PROTECTION_ ENABLED	SDOABORT_CODE_READ _PROTECTION_ENABLED	0F00 0013h
COMPARE_ERROR	SDOABORT_COMPARE_ERROR	0F00 000Ah
COUNT_ERROR	SDOABORT_COUNT_ERROR	0F00 0006h
DST_ADDR_ERROR	SDOABORT_DST_ADDR_ERROR	0F00 0003h
DST_ADDR_NOT_MAPPED	SDOABORT_DST_ADDR_ NOT_MAPPED	0F00 0005h
INVALID_CODE	SDOABORT_INVALID_CODE	0F00 0010h
INVALID_COMMAND	SDOABORT_INVALID_COMMAND	0F00 0001h
INVALID_SECTOR	SDOABORT_INVALID_SECTOR	0F00 0007h
PARAM_ERROR	SDOABORT_PARAM_ERROR	0F00 000Ch
SECTOR_NOT_BLANK	SDOABORT_SECTOR_ NOT_BLANK	0F00 0008h
SECTOR_NOT_PREPARED_ FOR_WRITE_OPERATION	SDOABORT_SECTOR_NOT_ PREPARED_FOR_WRITE_ OPERATION	0F00 0009h
SRC_ADDR_ERROR	SDOABORT_SRC_ADDR_ERROR	0F00 0002h
SRC_ADDR_NOT_MAPPED	SDOABORT_SRC_ADDR_ NOT_MAPPED	0F00 0004h

In addition, the regular CANopen SDO Abort Codes for invalid accesses to OD entries are also supported.

2.7 Differences to fully-compliant CANopen

While the bootloader uses the SDO communication protocol and the Object Dictionary data organiation method, it is not a fully CiA 301 standard compliant CANopen node. In particular, the following features are not available or different to the standard:

- No Network Management (NMT) message processing
- ~~No Bootup Message~~
- No Heartbeat Message, no entry 1017h
- Uses proprietary SDO Abort Codes to indicate device errors

- “Empty” SDO responses during SDO segmented download/write to the node are shortened to one data byte, rather than full eight data bytes as the standard describes. This to speed up the communication.

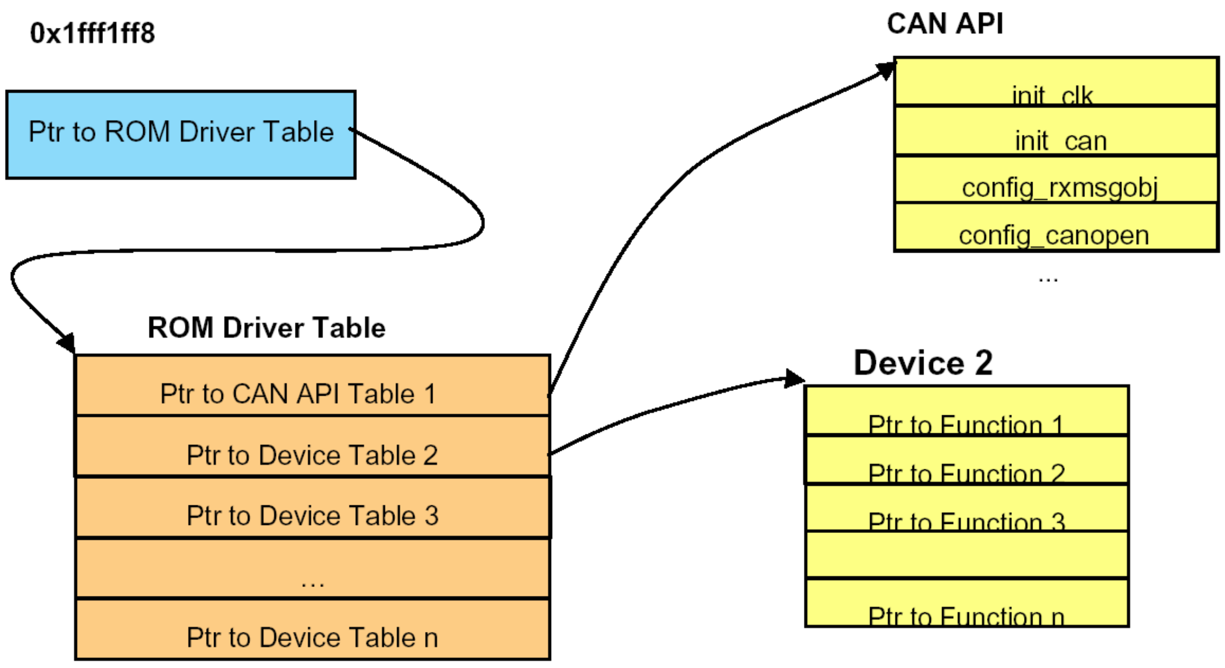
- Entry [1018h,1] Vendor ID reads 0000’0000h rather than an official CiA-assigned unique Vendor ID. This in particular because the chip will be incorporated into designs of customers who will become the “vendor” of the whole device. The host will have to use a different method to identify the CAN ISP devices.

Chapter 3 – CAN API

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 receive events.

3.1 Calling the CAN API

A fixed location in ROM contains a pointer to the ROM driver table i.e. 0x1fff1ff8. This location is kept the same for a device family. The ROM driver table contains pointer to the CAN API table. Pointers to the various CAN API functions are stored in this table. CAN API functions can be called by using a C structure. The figure below illustrates the pointer mechanism used to access the on-chip CAN API. On-chip RAM from address [START_CANAPI_RAM] to [END_API_RAM] is used by the CAN API. This address range should not be used by the application. For applications using the on-chip CAN API the linker control file should be modified appropriately to prevent usage of this area for application's variable storage.



In C, the structure with the function list that is referenced to call the API functions looks as follows:

```
typedef struct _CAND {
  void    (*init_can)        (uint32_t * can_cfg);
  void    (*isr)             (void);
  void    (*config_rxmsgobj) (CAN_MSG_OBJ * msg_obj);
  uint8_t (*can_receive)     (CAN_MSG_OBJ * msg_obj);
  void    (*can_transmit)    (CAN_MSG_OBJ * msg_obj);
  void    (*config_canopen)  (CAN_CANOPENCFG * canopen_cfg);
  void    (*canopen_handler) (void);
  void    (*config_calb)     (CAN_CALLBACKS * callback_cfg);
}  CAND;
```

Example that calls the CAN initialization:

```
ROM **rom = (ROM **)0x1fff1ff8;

uint32_t CanApiClkInitTable[2] = {
  0x00000000UL, // CANCLKDIV
  0x00004DC5UL // CAN_BTR
};

(*rom)->pCANAPI->init_can(&CanApiCanInitTable[0]);
```

3.2 CAN Initialization

The CAN controller clock divider is set and the CAN controller is initialized and its bitrate is set as well, based on an array of register values that are passed on via a pointer.

```
void init_can (uint32_t * can_cfg)
```

The first 32-bit value in the array is applied to the register CANCLKDIV, the second to CAN_BTR.

3.3 CAN Interrupt Handler

When the user application is active the interrupt handlers are mapped in the user flash space. The user application must provide an interrupt handler for the CAN interrupt. In order to process CAN events and call the callback functions the application must call the CAN API interrupt handler directly from the interrupt handler routine. The CAN API interrupt handler takes appropriate action as per the data received and status detected on the CAN bus.

```
void isr (void)
```

The CAN interrupt handler does **not** process CANopen messages.

Example call:

```
(*rom)->pCAND->isr();
```

3.4 CAN Rx Message Object Configuration

The CAN API supports and uses the full CAN model with 32 message objects. Any of the message objects can be used for receive or transmit of either 11-bit or 29-bit CAN messages. CAN messages that have their RTR-bit set (remote transmit) are also supported. For receive objects, a mask pattern for the message identifier allows to receive ranges of messages, up to receiving all CAN messages on the bus in a single message object. Transmit message objects are automatically configured when used.

```
// control bits for CAN_MSG_OBJ.mode_id
#define CAN_MSGOBJ_STD  0x00000000UL   // CAN 2.0a 11-bit ID
#define CAN_MSGOBJ_EXT  0x20000000UL   // CAN 2.0b 29-bit ID
#define CAN_MSGOBJ_DAT  0x00000000UL   // data frame
#define CAN_MSGOBJ_RTR  0x40000000UL   // rtr frame

typedef struct _CAN_MSG_OBJ {
  uint32_t  mode_id;
  uint32_t  mask;
  uint8_t   data[8];
  uint8_t   dlc;
  uint8_t   msgobj;
}  CAN_MSG_OBJ;


void config_rxmsgobj (CAN_MSG_OBJ * msg_obj)
```

Example call:

```
// Configure message object 1 to receive all 11-bit messages 0x000-0x00F
msg_obj.msgobj  = 1;
msg_obj.mode_id = 0x000;
msg_obj.mask    = 0x7F0;
(*rom)->pCAND-> config_rxmsgobj(&msg_obj);
```

3.5 CAN Receive

Reading messages that have been received by an Rx message object. A pointer to a message object structure is passed to the receive function. Before calling, the number of the message object that is to be read has to be set in the structure.

```
void config_rxmsgobj (CAN_MSG_OBJ * msg_obj)
```

Example call:

```
// Read out received message
msg_obj.msgobj = 5;
(*rom)->pCAND->can_receive(&msg_obj);
```

3.6 CAN Transmit

Setting up a message object and triggering the transmission of a CAN message on the bus. 11-bit standard and 29-bit extended messages are supported as well as both standard data and remote-transmit (RTR) messages.

```
void config_txmsgobj (CAN_MSG_OBJ * msg_obj)
```

Example call:

```
msg_obj.msgobj  = 3;
msg_obj.mode_id = 0x123UL;
msg_obj.mask    = 0x0UL;
msg_obj.dlc     = 1;
msg_obj.data[0] = 0x00;
(*rom)->pCAND->can_transmit(&msg_obj);
```

3.7 CANopen Configuration

The CAN API supports an Object Dictionary interface and the SDO protocol. In order to activate it, the CANopen configuration function has to be called with a pointer to a structure with the CANopen Node ID (1..127), the message object numbers to use for receive and transmit SDOs and two pointers to Object Dictionary configuration tables and their sizes. One table contains all read-only, constant entries of four bytes or less. The second table contains all variable and writable entries as well as SDO segmented entries.

```
typedef struct _CAN_ODCONSTENTRY {
  uint16_t index;
  uint8_t  subindex;
  uint8_t  len;
  uint32_t val;
}  CAN_ODCONSTENTRY;

// upper-nibble values for CAN_ODENTRY.entrytype_len
#define OD_NONE    0x00    // Object Dictionary entry doesn't exist
#define OD_EXP_RO  0x10    // Object Dictionary entry expedited, read-only
#define OD_EXP_WO  0x20    // Object Dictionary entry expedited, write-only
#define OD_EXP_RW  0x30    // Object Dictionary entry expedited, read-write
#define OD_SEG_RO  0x40    // Object Dictionary entry segmented, read-only
#define OD_SEG_WO  0x50    // Object Dictionary entry segmented, write-only
#define OD_SEG_RW  0x60    // Object Dictionary entry segmented, read-write

typedef struct _CAN_ODENTRY {
  uint16_t index;
  uint8_t  subindex;
  uint8_t  entrytype_len;
  uint8_t  *val;
}  CAN_ODENTRY;

typedef struct _CAN_CANOPENCFG {
  uint8_t   node_id;
  uint8_t   msgobj_rx;
  uint8_t   msgobj_tx;
  uint32_t  od_const_num;
  CAN_ODCONSTENTRY *od_const_table;
  uint32_t  od_num;
  CAN_ODENTRY *od_table;
}  CAN_CANOPENCFG;
```

Example OD tables and CANopen configuration structure:

```
// List of fixed, read-only Object Dictionary (OD) entries
// Expedited SDO only, length=1/2/4 bytes
const CAN_ODCONSTENTRY myConstOD [] = {
// index subindex length value
  { 0x1000, 0x00, 4, 0x54534554UL },  // "TEST"
  { 0x1018, 0x00, 1, 0x00000003UL },
  { 0x1018, 0x01, 4, 0x00000003UL },
  { 0x2000, 0x00, 1, (uint32_t)'M' },
};

// List of variable OD entries
// Expedited SDO with length=1/2/4 bytes
// Segmented SDO application-handled with length and value_pointer don't care
const CAN_ODENTRY myOD [] = {
// index subindex access_type|length value_pointer
  { 0x1001, 0x00, OD_EXP_RO | 1, (uint8_t *)&error_register },
  { 0x1018, 0x02, OD_EXP_RO | 4, (uint8_t *)&device_id },
  { 0x1018, 0x03, OD_EXP_RO | 4, (uint8_t *)&fw_ver },
  { 0x2001, 0x00, OD_EXP_RW | 2, (uint8_t *)&param },
  { 0x2200, 0x00, OD_SEG_RW,     (uint8_t *)NULL },
};

// CANopen configuration structure
const CAN_CANOPENCFG myCANopen = {
  20,  //   node_id
   5,  //   msgobj_rx
   6,  //   msgobj_tx
  sizeof(myConstOD)/sizeof(myConstOD[0]),  // od_const_num
  (CAN_ODCONSTENTRY *)myConstOD,           // od_const_table
  sizeof(myOD)/sizeof(myOD[0]),            // od_num
  (CAN_ODENTRY *)myOD,                     // od_table
};
```

Example call:

```
// Initialize CANopen handler
(*rom)->pCAND->config_canopen((CAN_CANOPENCFG *)&myCANopen);
```

3.8 CANopen Handler

The CANopen handler processes the CANopen SDO messages to access the Object Dictionary and calls the CANopen callback functions, when initialized. It has to be called cyclically as often as needed for the application.

In a CANopen application, SDO handling typically has the lowest priority and it is done in the foreground rather than interrupt processing.

Example call:

```
// Call CANopen handler
(*rom)->pCAND->canopen_handler();
```

3.9 CAN/CANopen Callback Functions

The CAN API supports callback functions for various events. The callback functions are published via an API function.

```
typedef struct _CAN_CALLBACKS {
  void (*CAN_rx)(uint8_t msg_obj);
  void (*CAN_tx)(uint8_t msg_obj);
  void (*CAN_error)(uint32_t error_info);
  uint32_t (*CANOPEN_sdo_read)(uint16_t index, uint8_t subindex);
  uint32_t (*CANOPEN_sdo_write)(
    uint16_t index, uint8_t subindex, uint8_t *dat_ptr);
  uint32_t (*CANOPEN_sdo_seg_read)(
    uint16_t index, uint8_t subindex, uint8_t openclose,
    uint8_t *length, uint8_t *data, uint8_t *last);
  uint32_t (*CANOPEN_sdo_seg_write)(
    uint16_t index, uint8_t subindex, uint8_t openclose,
    uint8_t length, uint8_t *data, uint8_t *fast_resp);
  uint8_t (*CANOPEN_sdo_req)(
    uint8_t length_req, uint8_t *req_ptr, uint8_t *length_resp,
    uint8_t *resp_ptr);
}  CAN_CALLBACKS;
```

Example callback table definition:

```
// List of callback function pointers
const CAN_CALLBACKS callbacks = {
  CAN_rx,
  CAN_tx,
  CAN_error,
  CANOPEN_sdo_exp_read,
  CANOPEN_sdo_exp_write,
  CANOPEN_sdo_seg_read,
  CANOPEN_sdo_seg_write,
  CANOPEN_sdo_req,
};
```

Example call:

```
// Publish callbacks
(*rom)->pCAND->config_calb((CAN_CALLBACKS *)&callbacks);
```

3.9.1 CAN Message Received Callback

Called on the interrupt level by the CAN interrupt handler.

Example:

```
// CAN receive handler
void CAN_rx(uint8_t msgobj_num)
{
  // Read out received message
  msg_obj.msgobj = msgobj_num;
  (*rom)->pCAND->can_receive(&msg_obj);

  return;
}
```

Note: The callback is not called if the user CANopen handler is activated, for the message object that is used for SDO receive.

3.9.2 CAN Message Transmit Callback

Called on the interrupt level by the CAN interrupt handler. The callback function is called before the SDO response is generated, allowing to modify or update the data.

Example:

```
// CAN transmit handler
void CAN_tx(uint8_t msgobj_num)
{
  // Reset flag used by application to wait for transmission finished
  if (wait_for_tx_finished == msgobj_num)
    wait_for_tx_finished = 0;

  return;
}
```

Note: The callback is not called if the user CANopen handler is activated, for the message object that is used for SDO transmit.

3.9.3 CAN Error Callback

Called on the interrupt level by the CAN interrupt handler.

```
// error status bits
#define CAN_ERROR_NONE  0x00000000UL
#define CAN_ERROR_PASS  0x00000001UL
#define CAN_ERROR_WARN  0x00000002UL
#define CAN_ERROR_BOFF  0x00000004UL
#define CAN_ERROR_STUF  0x00000008UL
#define CAN_ERROR_FORM  0x00000010UL
#define CAN_ERROR_ACK   0x00000020UL
#define CAN_ERROR_BIT1  0x00000040UL
#define CAN_ERROR_BIT0  0x00000080UL
#define CAN_ERROR_CRC   0x00000100UL
```

Example:

```
// CAN error handler
```

```
void CAN_error(uint32_t error_info)
{
  // If we went into bus off state, tell the application to
  // re-initialize the CAN controller
  if (error_info & CAN_ERROR_BOFF)
    reset_can = TRUE;

  return;
}
```

3.9.4 CANopen SDO Expedited Read Callback

Called by the CANopen handler. The callback function is called before the SDO response is generated, allowing to modify or update the data.

Example:

```
// CANopen callback for expedited read accesses
uint32_t CANOPEN_sdo_exp_read(uint16_t index, uint8_t subindex)
{
  // Every read of [2001h,0] increses param by one
  if ((index == 0x2001) && (subindex==0))
    param++;

  return 0;
}
```

3.9.5 CANopen SDO Expedited Write Callback

Called by the CANopen handler. The callback passes on the new data and is called before the new data has been written, allowing to reject or condition the data.

Example:

```
// CANopen callback for expedited write accesses
uint32_t CANOPEN_sdo_exp_write(uint16_t index, uint8_t subindex, uint8_t
*dat_ptr)
{
  // Writing 0xAA55 to entry [2001h,0] unlocks writing the config table
  if ((index == 0x2001) && (subindex == 0))
    if (*(uint16_t *)dat_ptr == 0xAA55)
    {
      write_config_ena = TRUE;
      return(TRUE);
    }
    else
      return(FALSE);  // Reject any other value
}
```

3.9.6 CANopen SDO Segmented Read Callback

Called by the CANopen handler. The callback informs about the opening of the read channel and allows to provide data segments of up to seven bytes to the reading host. It also allows to close the channel when all data has been read or to abort the transmission at any time.

```
// Values for CANOPEN_sdo_seg_read/write() callback 'openclose' parameter
#define CAN_SDOSEG_SEGMENT        0  // segment read/write
#define CAN_SDOSEG_OPEN           1  // channel is opened
#define CAN_SDOSEG_CLOSE          2  // channel is closed
```

Example for reading a buffer:

```
uint8_t read_buffer[0x123];


// CANopen callback for segmented read accesses
uint32_t CANOPEN_sdo_seg_read(
  uint16_t index, uint8_t subindex, uint8_t openclose,
  uint8_t *length, uint8_t *data, uint8_t *last)
{
  static uint16_t read_ofs;
  uint16_t i;

  if ((index == 0x2200) && (subindex==0))
  {
    if (openclose == CAN_SDOSEG_OPEN)
    {
      // Initialize the read buffer with "something"
      for (i=0; i<sizeof(read_buffer); i++)
      {
        read_buffer[i] = (i+5) + (i<<2);
      }
      read_ofs = 0;
    }
    else if (openclose == CAN_SDOSEG_SEGMENT)
    {
      i = 7;
      while (i && (read_ofs < sizeof(read_buffer)))
      {
        *data++ = read_buffer[read_ofs++];
        i--;
      }
      *length = 7-i;
      if (read_ofs == sizeof(read_buffer)) // The whole buffer read:
                                           // this is last segment
      {
        *last = TRUE;
      }
    }
    return 0;
  }
  else
  {
    return SDO_ABORT_NOT_EXISTS;
  }
}
```

3.9.7 CANopen SDO Segmented Write Callback

Called by the CANopen handler. The callback informs about the opening and closing of the write channel and passes on data segments of up to seven bytes from the writing host. It also allows to abort the transmission at any time, for example when there is a buffer overflow. Responses can be selected to be 8-byte (CANopen standard compliant) or 1-byte (faster but not supported by all SDO clients).

```
// Values for CANOPEN_sdo_seg_read/write() callback 'openclose' parameter
#define CAN_SDOSEG_SEGMENT        0  // segment read/write
#define CAN_SDOSEG_OPEN           1  // channel is opened
#define CAN_SDOSEG_CLOSE          2  // channel is closed
```

Example for writing a buffer:

```
uint8_t write_buffer[0x321];


// CANopen callback for segmented write accesses
uint32_t CANOPEN_sdo_seg_write(
  uint16_t index, uint8_t subindex, uint8_t openclose,
  uint8_t length, uint8_t *data, uint8_t *fast_resp)
{
  static uint16_t write_ofs;
  uint16_t i;

  if ((index == 0x2200) && (subindex==0))
  {
    if (openclose == CAN_SDOSEG_OPEN)
    {
      // Initialize the write buffer
      for (i=0; i<sizeof(write_buffer); i++)
      {
        write_buffer[i] = 0;
      }
      write_ofs = 0;
    }
    else if (openclose == CAN_SDOSEG_SEGMENT)
    {
      *fast_resp = TRUE;  // Use fast 1-byte segment write response
      i = length;
      while (i && (write_ofs < sizeof(write_buffer)))
      {
        write_buffer[write_ofs++] = *data++;
        i--;
      }
      if (i && (write_ofs >= sizeof(write_buffer))) // Too much data to write
      {
        return SDO_ABORT_TRANSFER;  // Data could not be written
      }
    }
    else if (openclose == CAN_SDOSEG_CLOSE)
    {
      // Write has successfully finished: mark the buffer valid etc.
    }
    return 0;
  }
  else
  {
    return SDO_ABORT_NOT_EXISTS;
  }
}
```

3.9.8 CANopen Fall-Back SDO Handler Callback

Called by the CANopen handler. It is called whenever an SDO request could not be processed or would end in an SDO abort response. It is called with the full data buffer of the request and allows to generate any type of SDO response. This can be used to implement custom SDO handlers, for example to implement the SDO block transfer method.

```
// Return values for CANOPEN_sdo_req() callback
#define CAN_SDOREQ_NOTHANDLED     0  // process regularly, no impact
#define CAN_SDOREQ_HANDLED_SEND   1  // processed in callback, auto-send
                                     // returned msg
#define CAN_SDOREQ_HANDLED_NOSEND 2  // processed in callback, don't send
                                     // response
```

Example (not implementing custom processing):

```
// CANopen callback for custom SDO request handler
uint8_t CANOPEN_sdo_req (
  uint8_t length, uint8_t *req_ptr, uint8_t *length_resp, uint8_t *resp_ptr)
{
  return CAN_SDOREQ_NOTHANDLED;
}
```