AN13121

Porting CANopennode stack on LPC5500 series

Rev. 0 — 29 January 2021

Application Note

Contents

1 CANopen overview

CANopen is a CAN-based communication system. It comprises higher-layer protocols and profile specifications. CANopen has been developed as a standardized embedded network with highly flexible configuration capabilities. Today it is used in various application fields, such as medical equipment, off-road vehicles, maritime electronics, railway applications, or building automation. This application note assumes that the reader already has a basic knowledge of CAN2.0A protocol, CAN frame structure, ID assignment, data filed, and DLC.

A General CANopen device has three logical parts as in Figure 1.

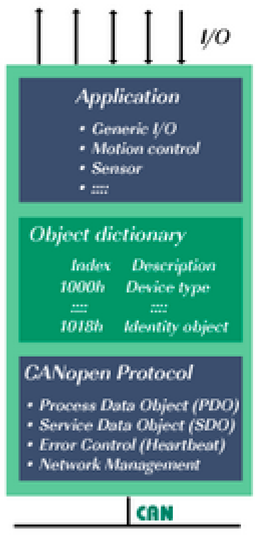


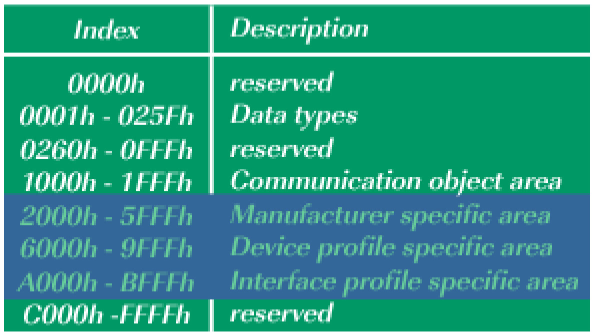
Figure 1. CANopen system

1. **CANopen protocol stack** handles the communication via the CAN network.
2. **Application software** provides the internal control functionality as well as the interface to the process hardware interfaces.
3. **Object dictionary** interfaces the protocol as well as the application software. It contains references (indices) for all used data types and stores all communication and application parameters.

1.1 Object dictionary

Object dictionary for a CANopen device is like registers for a microcontroller, which is the most important part in a CANopen device. All communication and application parameters that determine the behavior of a CANopen device organized in a serialized table named object dictionary.

In this table, all data that must be exchanged between the communication part (CANopen protocol stack) and the application part (application software) of the device owns unique 24-bit addresses. This address is divided into a 16-bit index and an 8-bit subindex. Any entry in the CANopen object dictionary is readable via CANopen communication services. Different indices in an OB (Object dictionary) are predefined into sections targeting different usage.



Index	Description
0000h	reserved
0001h - 025Fh	Data types
0260h - 0FFFh	reserved
1000h - 1FFFh	Communication object area
2000h - 5FFFh	Manufacturer specific area
6000h - 9FFFh	Device profile specific area
A000h - BFFFh	Interface profile specific area
C000h -FFFFh	reserved

Figure 2. Object dictionary

1.2 Summary

The sections above are the basic concepts of CANopen. If you are new to CANopen and need details, see CiA (CAN in automation) website https://www.can-cia.org/canopen/. There is a plenty of useful learning material and example about CANopen. You may download the Application Layer and Communication Profile, Standard 301, which is the core protocol specification of CANopen and is freely available for download on CiA website.

2 LPC5500 series CAN and SDK driver

This section describes:

- MCAN
- SDK driver
- SDK example

2.1 MCAN

The LPC55S16/LPC55S06 features a powerful CAN controller. The LPC55S16 SDK provides API wrapper to help customer easily use MCAN. This section focuses on SDK MCAN driver usage as it is a prerequisite for porting CANopen stack on to LPC5500.

The MCAN block is available on the LPC55S1x/LPC55S0x devices. It conforms to CAN protocol version2.0 Part A, B definition, and support CANFD frame. It has two configurable receive FIFOs, up to 64 dedicated received buffer, and up to 32 dedicated transmit buffers. See Figure 3 for MCAN block diagram on LPC5500 series.

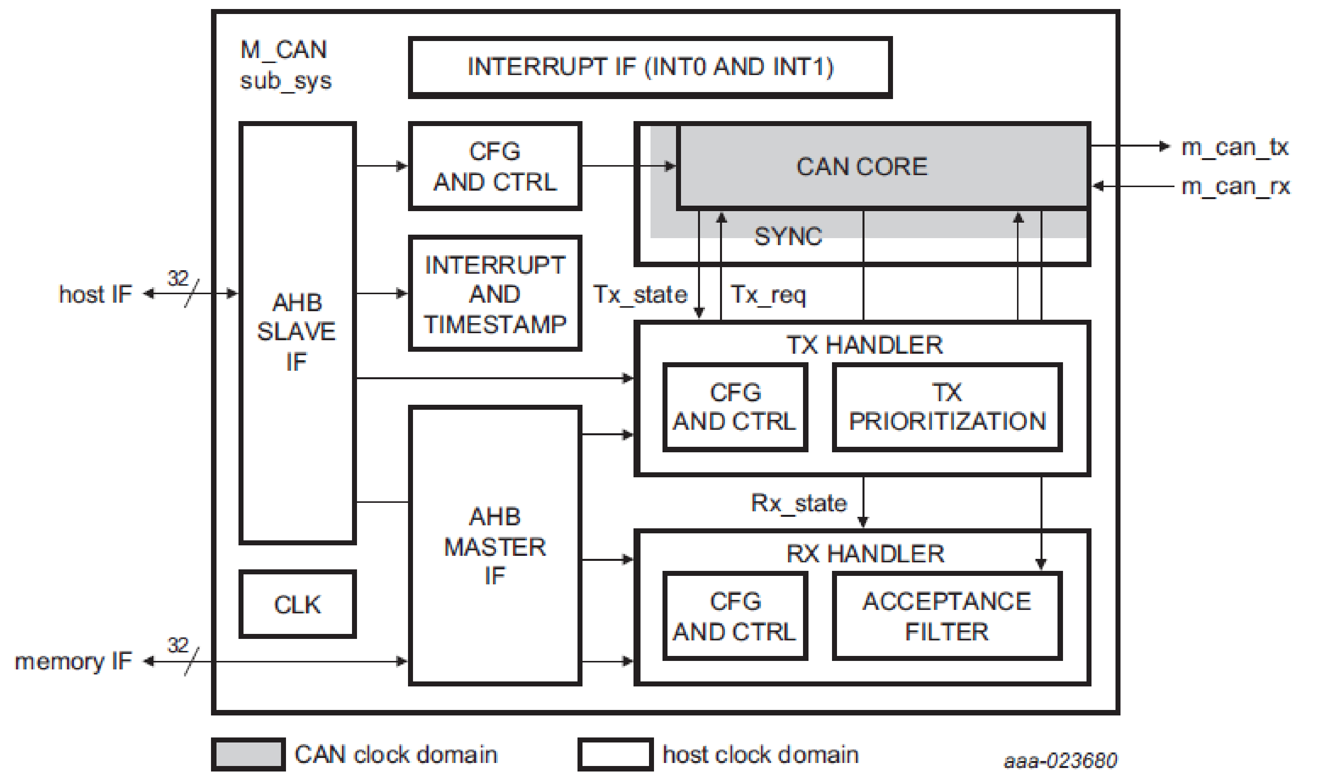


Figure 3. MCAN block diagram

One of the key concepts on MCAN configuration is message RAM configuration. The MCAN module occupies and allocates up to 4352 words in message RAM. The message RAM is a normal block in the system SRAM. It is divided into several subregion and each region has its functionality, such as TX FIFO, RX FIFO, or filter template. You can use SDK driver to configure the subregion size. For details, see Figure 4.

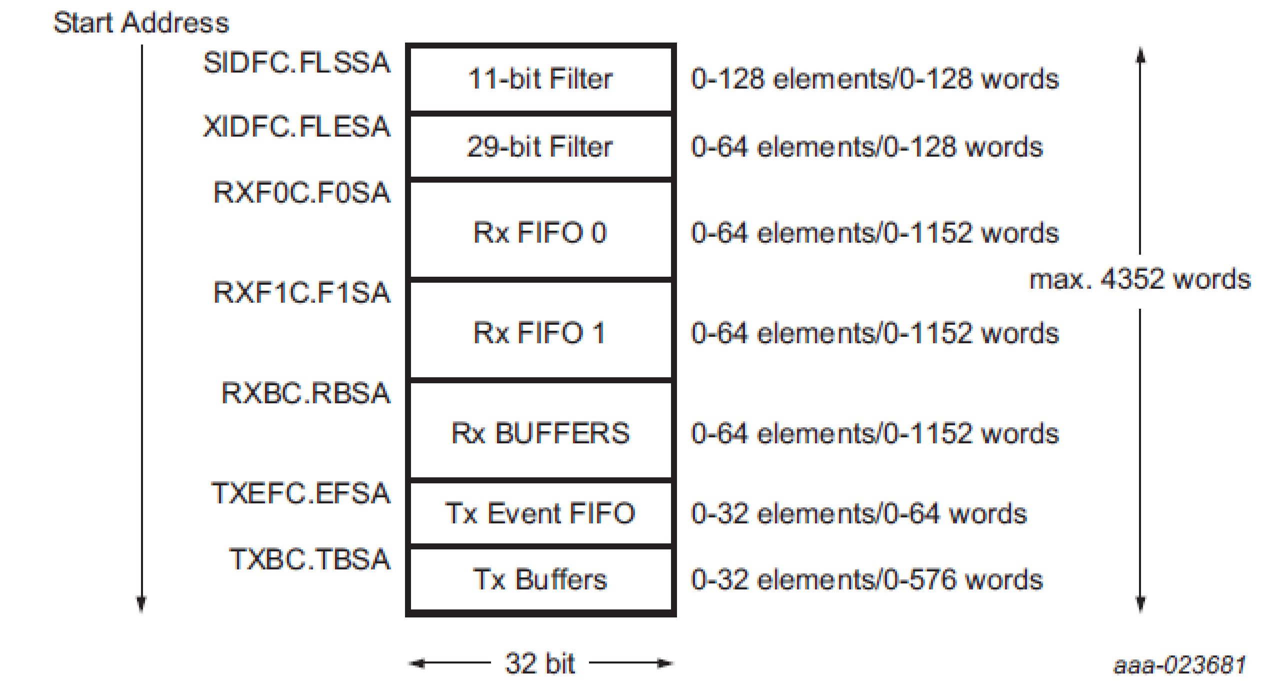


Figure 4. Message RAM configuration

2.2 SDK driver

MCUXpresso SDK provides a full set of APIs to cover all the functions of MCAN module. MCUXpresso SDK also provides two dedicated examples for demonstration:

MCAN driver file location:

- SDK_2.8.0_LPCXpresso55S16\devices\LPC55S16\drivers\fsl_mcan.c
- SDK_2.8.0_LPCXpresso55S16\devices\LPC55S16\drivers\fsl_mcan.h

Examples location:

- SDK_2.8.0_LPCXpresso55S16\boards\lpcxpresso55s16\driver_examples\mcan

2.3 SDK example

The SDK example **interrupt_transfer** is the best starting point to play with MCAN.

The steps for using SDK driver for basic CAN communication are:

1. Configure and enable MCAN input clock, MCAN functional clock is from main_clock.

```
/* Set MCAN clock 100Mhz/5=20MHz. */
CLOCK_SetClkDiv(kCLOCK_DivCanClk, 5U, true);
CLOCK_AttachClk(kMCAN_DIV_to_MCAN);
```

2. Initialize the MCAN module - Set the initialization structure to default value and call MCAN_Init to initialize the MCAN module.

```
MCAN_GetDefaultConfig(&mcanConfig);
MCAN_Init(EXAMPLE_MCAN, &mcanConfig, MCAN_CLK_FREQ);
```

3. RX filter configuration - Enable RX filter and let specific RX frame pass through filter, reject all other CAN frames.

   ```
   /* STD filter config. */
   rxFilter.address  = STD_FILTER_OFS;
   rxFilter.idFormat = kMCAN_FrameIDStandard;
   rxFilter.listSize = 1U;
   rxFilter.nmFrame  = kMCAN_reject0;
   rxFilter.remFrame = kMCAN_rejectFrame;
   MCAN_SetFilterConfig(EXAMPLE_MCAN, &rxFilter);

   stdFilter.sfec = kMCAN_storeinFifo0;
   /* Classic filter mode, only filter matching ID. */
   stdFilter.sft= kMCAN_classic;
   stdFilter.sfid1 = rxIdentifier;
   stdFilter.sfid2 = 0x7FFU;
   MCAN_SetSTDFilterElement(EXAMPLE_MCAN, &rxFilter, &stdFilter, 0);
   ```

4. RX FIFO and TX buffer configuration, setting TX buffer and RX FIFO to be classic CAN format and assign the memory into txBuffer.address filed.

   ```
   /* RX fifo0 config. */
   rxFifo0.address = RX_FIFO0_OFS;
   rxFifo0.elementSize= 1U;
   rxFifo0.watermark  = 0;
   rxFifo0.opmode  = kMCAN_FifoBlocking;
   rxFifo0.datafieldSize = kMCAN_8ByteDatafield;
   MCAN_SetRxFifo0Config(EXAMPLE_MCAN, &rxFifo0);

   /* TX buffer config. */
   memset(&txBuffer, 0, sizeof(txBuffer));
   txBuffer.address = TX_BUFFER_OFS;
   txBuffer.dedicatedSize = 1U;
   txBuffer.fqSize  = 0;
   txBuffer.datafieldSize = kMCAN_8ByteDatafield;
   MCAN_SetTxBufferConfig(EXAMPLE_MCAN, &txBuffer);
   ```

5. Set MCAN module into operation mode and start to send/receive CAN frame.

   ```
   MCAN_EnterNormalMode(EXAMPLE_MCAN);
   ```

   a. Send CAN frames - Set all fields in a txFrame structure and call MCAN_TransferSendNonBlocking to send out CAN frames.

      ```
      /* Config TX frame data. */
      memset(tx_data, 0, sizeof(uint8_t) * CAN_DATASIZE);
      for (cnt = 0; cnt < CAN_DATASIZE; cnt++)
      {
       tx_data[cnt] = cnt;
      }
      tx_data[0] += numMessage++;
      txFrame.xtd  = kMCAN_FrameIDStandard;
      txFrame.rtr  = kMCAN_FrameTypeData;
      txFrame.fdf  = 0;
      txFrame.brs  = 0;
      txFrame.dlc  = 8U;
      txFrame.id= txIdentifier << STDID_OFFSET;
      txFrame.data = tx_data;
      txFrame.size = CAN_DATASIZE;
      txXfer.frame  = &txFrame;
      ```

```
txXfer.bufferIdx = 0;
MCAN_TransferSendNonBlocking(EXAMPLE_MCAN, &mcanHandle, &txXfer);
```

b. Receive a CAN frame: SDK use callback function to notify application layer a CAN frame has been received, user application should mark a flag in can_callback function and copy the received frame and process it in the main thread.

```
static void mcan_callback(CAN_Type *base, mcan_handle_t *handle, status_t status, uint32_t
result, void *userData)
{
 switch (status)
 {
  case kStatus_MCAN_RxFifo0Idle:
  {
rxComplete = true;
  }
  break;

  case kStatus_MCAN_TxIdle:
  {
txComplete = true;
  }
  break;

  default:
break;
 }
}
```

3 Porting CANopenNode on LPC5500 series

This section describes:

- CANopenNode
- Porting CANopenNode
- Test

3.1 CANopenNode

CANopenNode is free and open source CANopen protocol stack written in ANSI C. It can run on different microcontrollers, as standalone application or with RTOS. Variables (communication, device, custom) are ordered in CANopen Object Dictionary and are accessible from both: C code and from CANopen network.

Github page for CANopenNode project is : https://github.com/CANopenNode/CANopenNode.

CANopenNode support following CANopen features:

- Heartbeat producer/consumer error control.
- PDO linking and dynamic mapping for fast exchange of process variables.
- SDO expedited, segmented and block transfer for service access to all parameters.
- SDO master.
- Emergency message.
- Sync producer/consumer.
- Time protocol (producer/consumer).
- Non-volatile storage.

3.2 Porting CANopenNode

CANopenNode itself does not have complete working code for any microcontroller. It is only the library with the stack. It has example, which should compile on any system with template driver (drvTemplate), which actually does not access CAN hardware.

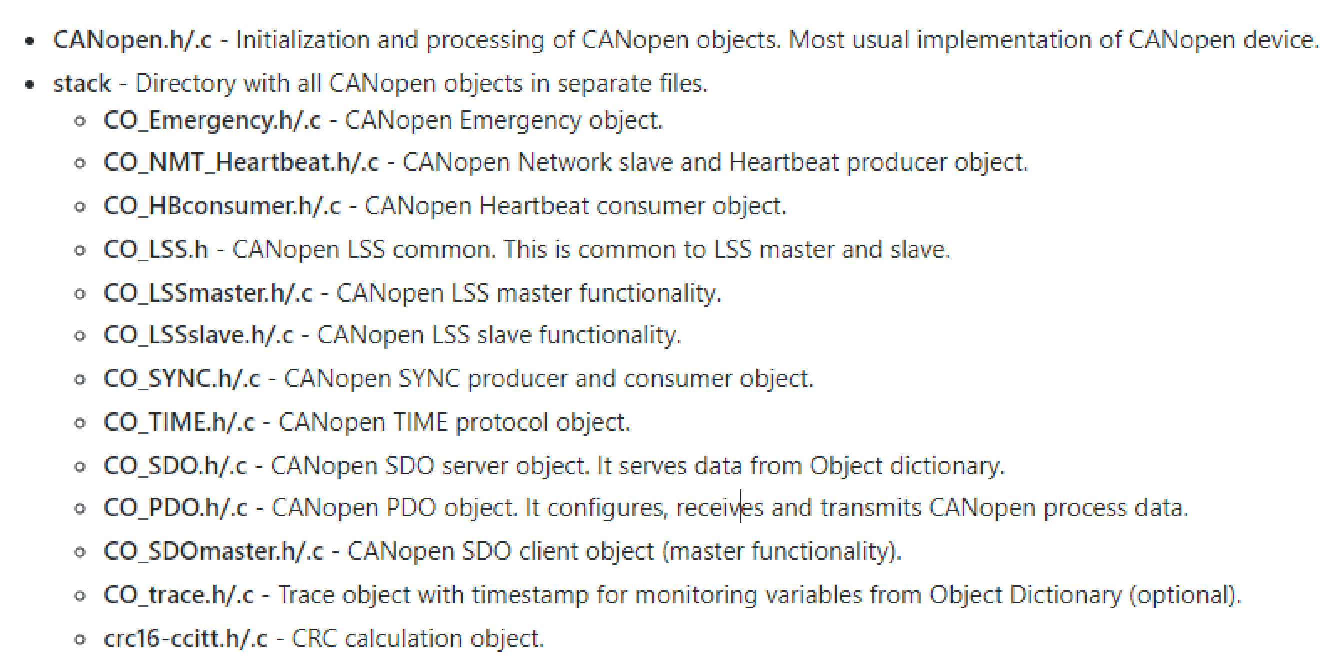
3.2.1 Download source code

We use CANopenNodeV1.3, so the default download page is not correct, please download source code on:

https://github.com/CANopenNode/CANopenNode/tree/v1.3-master

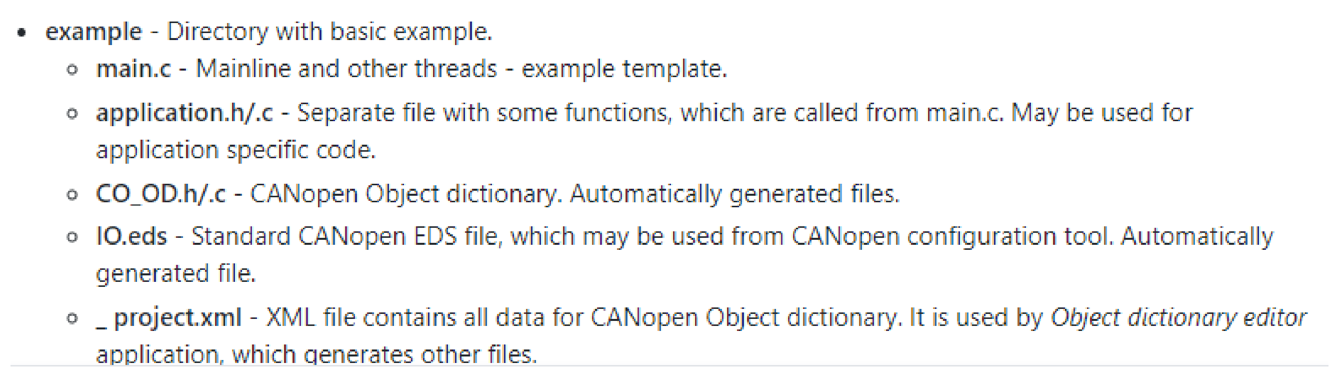
3.2.2 File structure

The stack file structure shown in Figure 5 is not modified.



- **CANopen.h/.c** - Initialization and processing of CANopen objects. Most usual implementation of CANopen device.
- **stack** - Directory with all CANopen objects in separate files.
  - **CO_Emergency.h/.c** - CANopen Emergency object.
  - **CO_NMT_Heartbeat.h/.c** - CANopen Network slave and Heartbeat producer object.
  - **CO_HBconsumer.h/.c** - CANopen Heartbeat consumer object.
  - **CO_LSS.h** - CANopen LSS common. This is common to LSS master and slave.
  - **CO_LSSmaster.h/.c** - CANopen LSS master functionality.
  - **CO_LSSslave.h/.c** - CANopen LSS slave functionality.
  - **CO_SYNC.h/.c** - CANopen SYNC producer and consumer object.
  - **CO_TIME.h/.c** - CANopen TIME protocol object.
  - **CO_SDO.h/.c** - CANopen SDO server object. It serves data from Object dictionary.
  - **CO_PDO.h/.c** - CANopen PDO object. It configures, receives and transmits CANopen process data.
  - **CO_SDOmaster.h/.c** - CANopen SDO client object (master functionality).
  - **CO_trace.h/.c** - Trace object with timestamp for monitoring variables from Object Dictionary (optional).
  - **crc16-ccitt.h/.c** - CRC calculation object.

Figure 5. CANopenNode stack file structure



- **example** - Directory with basic example.
  - **main.c** - Mainline and other threads - example template.
  - **application.h/.c** - Separate file with some functions, which are called from main.c. May be used for application specific code.
  - **CO_OD.h/.c** - CANopen Object dictionary. Automatically generated files.
  - **IO.eds** - Standard CANopen EDS file, which may be used from CANopen configuration tool. Automatically generated file.
  - **_project.xml** - XML file contains all data for CANopen Object dictionary. It is used by *Object dictionary editor* application, which generates other files.

Figure 6. CANopenNode example file structure

3.2.3 Flowchart of a typical CANopenNode implementation

CANopenNode need three threads to operate. as shown in Figure 7.

```
                   -----------------------
                  |     Program start     |
                   -----------------------
                              |
                   -----------------------
                  |     CANopen init      |
                   -----------------------
                              |
                   -----------------------
                  |     Start threads     |
                   -----------------------
                        |     |     |
     --------------------     |     --------------------
    |                         |                         |
 -----------------------   -----------------------   -----------------------
| CAN receive thread    | | Timer interval thread | | Mainline thread       |
|                       | |                       | |                       |
| - Fast response.      | | - Realtime thread with| | - Processing of time  |
| - Detect CAN ID.      | |   constant interval,  | |   consuming tasks     |
| - Partially process   | |   typically 1ms.      | |   in CANopen objects: |
|   messages and copy   | | - Network synchronized| |    - SDO server,      |
|   data to target      | | - Copy inputs (RPDOs, | |    - Emergency,       |
|   CANopen objects.    | |   HW) to Object Dict. | |    - Network state,   |
|                       | | - May call application| |    - Heartbeat.       |
|                       | |   for some processing.| | - May cyclically call |
|                       | | - Copy variables from | |   application code.   |
|                       | |   Object Dictionary to| |                       |
|                       | |   outputs (TPDOs, HW).| |                       |
 -----------------------   -----------------------   -----------------------
```

Figure 7. CANopenNode stack file structure

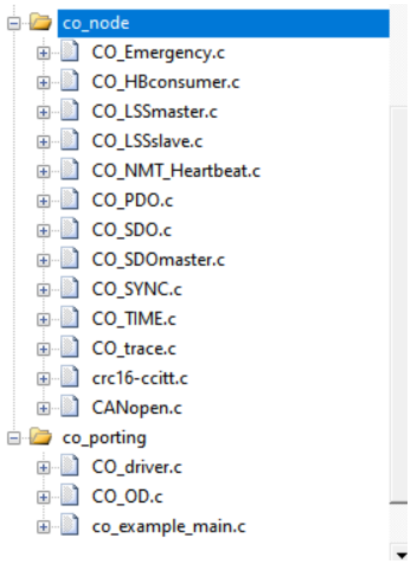
Here the focus is on BareMetal porting, which is simplified into main loop and interrupt handling:

1. The mainline thread is put into while(1) loop.
2. The Timer interval thread need to be called every 1ms, we put timer interval thread into a 1ms interrupt services handler.
3. Put the CAN receive thread into SDK's callback handler function, which in essence handled in hardware interrupt handler.

It is necessary to protect critical sections, where different threads access to the same resource. In simple BareMetal interrupts systems: user may be need temporary disabled between access to the shared resource.

3.2.4 Implement CO_Driver

Put stack source code and example file into your project, include the corresponding include path, as Fig shows:



co_node
CO_Emergency.c
CO_HBconsumer.c
CO_LSSmaster.c
CO_LSSslave.c
CO_NMT_Heartbeat.c
CO_PDO.c
CO_SDO.c
CO_SDOmaster.c
CO_SYNC.c
CO_TIME.c
CO_trace.c
crc16-ccitt.c
CANopen.c
co_porting
CO_driver.c
CO_OD.c
co_example_main.c

Figure 8. Add CANopenNode stack into project

The remaining task focus on implement CO_driver.c to connect SDK's MCAN driver to CANopenNode stack.

The CO_driver.c/h is responsible for:

- Basic data types.
- Receive and transmit buffers for CANopen messages.
- Interaction with CAN module on the microcontroller.
- CAN receive and transmit interrupts.

The critical function needs to be implemented are list as below:

Table 1. Critical functions

Function	Description
CO_CANrxBufferInit	Configures specific CAN receive buffer. It sets CAN identifier and connects buffer with specific object.
CO_CANtxBufferInit	Configures specific CAN transmit buffer.
CO_CANsend	Send CAN message.

CO_CANsend: This is the easiest function to implementation, just set the txFrame filed and call *MCAN_TransferSendNonBlocking*.

```
static mcan_buffer_transfer_t txXfer;
static mcan_tx_buffer_frame_t txFrame;
txFrame.xtd = kMCAN_FrameIDStandard;
txFrame.rtr = kMCAN_FrameTypeData;
txFrame.fdf = 0;
txFrame.brs = 0;
txFrame.dlc = len;
txFrame.id = id << STDID_OFFSET;
txFrame.data = buf;
txFrame.size = len;
txXfer.frame = &txFrame;
```

```
 txXfer.bufferIdx = 0;
 MCAN_TransferSendNonBlocking(CAN0, &mcanHandle, &txXfer);
```

CO_CANtxBufferInit: CO_CANtxBufferInit() returns a pointer of type CO_CANtx_t, which contains buffer where CAN message data can be written. CAN message is send with calling CO_CANsend() function. If at that moment CAN transmit buffer inside microcontroller's CAN module is free, message is copied directly to CAN module.

```
 CO_CANtx_t *buffer = NULL;
 buffer = &CANmodule->txArray[0];
 if((CANmodule != NULL) && (index < CANmodule->txSize)){
 /* get specific buffer */
 buffer = &CANmodule->txArray[index];
 /* CAN identifier, DLC and rtr, bit aligned with CAN module transmit buffer. * Microcontroller
specific. */
 buffer->ident = ((uint32_t)ident & 0x07FFU)
 | ((uint32_t)(((uint32_t)noOfBytes & 0xFU) << 12U))
 | ((uint32_t)(rtr ? 0x8000U : 0U));
 buffer->bufferFull = false;
 buffer->syncFlag = syncFlag;
 buffer->DLC = noOfBytes;
 }
 return buffer;
```

CO_CANrxBufferInit: Before CAN messages can be received, each member in CO_CANrx_t must be initialized. CO_CANrxBufferInit() is called by CANopen module, which uses specific member. Main arguments to the CO_CANrxBufferInit() function are CAN identifier and a pointer to callback function. Those two arguments (and some others) are copied to the member of the CO_CANrx_t array.

```
CO_ReturnError_t ret = CO_ERROR_NO;
 if((CANmodule!=NULL) && (object!=NULL) && (pFunct!=NULL) && (index < CANmodule->rxSize)){
 /* buffer, which will be configured */
 CO_CANrx_t *buffer = &CANmodule->rxArray[index];
 /* Configure object variables */
 buffer->object = object;
 buffer->pFunct = pFunct;
 /* CAN identifier and CAN mask, bit aligned with CAN module. Different on different
microcontrollers. */
 buffer->ident = ident & 0x07FFU;
 if(rtr){
 buffer->ident |= 0x0800U;
 }
 buffer->mask = (mask & 0x07FFU) | 0x0800U;
 }
 else{
 ret = CO_ERROR_ILLEGAL_ARGUMENT;
 }
 return ret;
```

The CAN frame receive processing is completed in SDK’s MCAN callback. Every time the hardware receives a CAN frame, the software checks if the CAN frame id matches to any of the rxArray ID. If match occurs, software copies the CAN message and calls CANopen’s msgbuf’s callback function for further processing.

```
static void mcan_callback(CAN_Type *base, mcan_handle_t *handle, status_t status, uint32_t result,
void *userData)
{
 int i;
 if(status == kStatus_MCAN_RxFifo0Idle)
 {
 CO_CANrxMsg_t rcvMsg;
 rcvMsg.DLC = rxFrame.size;
 memcpy(rcvMsg.data, rxFrame.data, rcvMsg.DLC);
 rcvMsg.ident = rxFrame.id >> STDID_OFFSET;
 CO_CANrx_t *msgBuff = NULL;
 for(i=0; i<CO->CANmodule[0]->rxSize; i++)
 {
 if( CO->CANmodule[0]->rxArray[i].ident == rcvMsg.ident)
 {
 msgBuff = &CO->CANmodule[0]->rxArray[i];
 }
 }
 if (msgBuff && msgBuff->pFunct)
 {
 msgBuff->pFunct(msgBuff->object, &rcvMsg);
 }
 MCAN_TransferReceiveFifoNonBlocking(CAN0, 0, &mcanHandle, &rxXfer);
 }
 if(status == kStatus_MCAN_TxIdle)
 {
 }
}
```

3.3 Test

This section describes the Hardware environment and the Test Step.

3.3.1 Hardware environment

- Boards
  - One LPCXpresso55S16 boards
  - One USB-CAN bridge convertor
- Miscellaneous
  - One Micro USB cables
  - One 120 Ohm terminated CAN cable
  - Personal Computer
- Boards Setup

There is no specific jumper setting for LPC55S16, please follow SDK example: mcan_interrupt's readme file for CAN jumper setting. The block diagram is shown in Figure 9 and Figure 10.

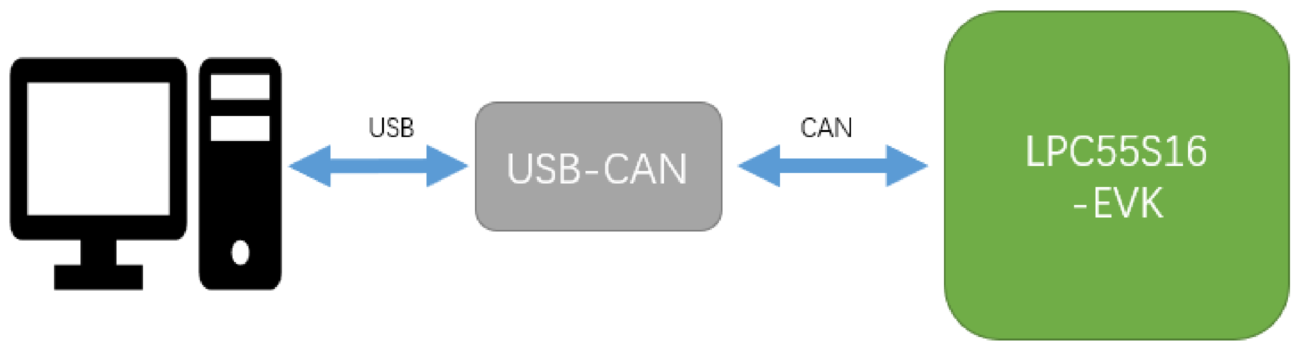


Figure 9. Message RAM configuration

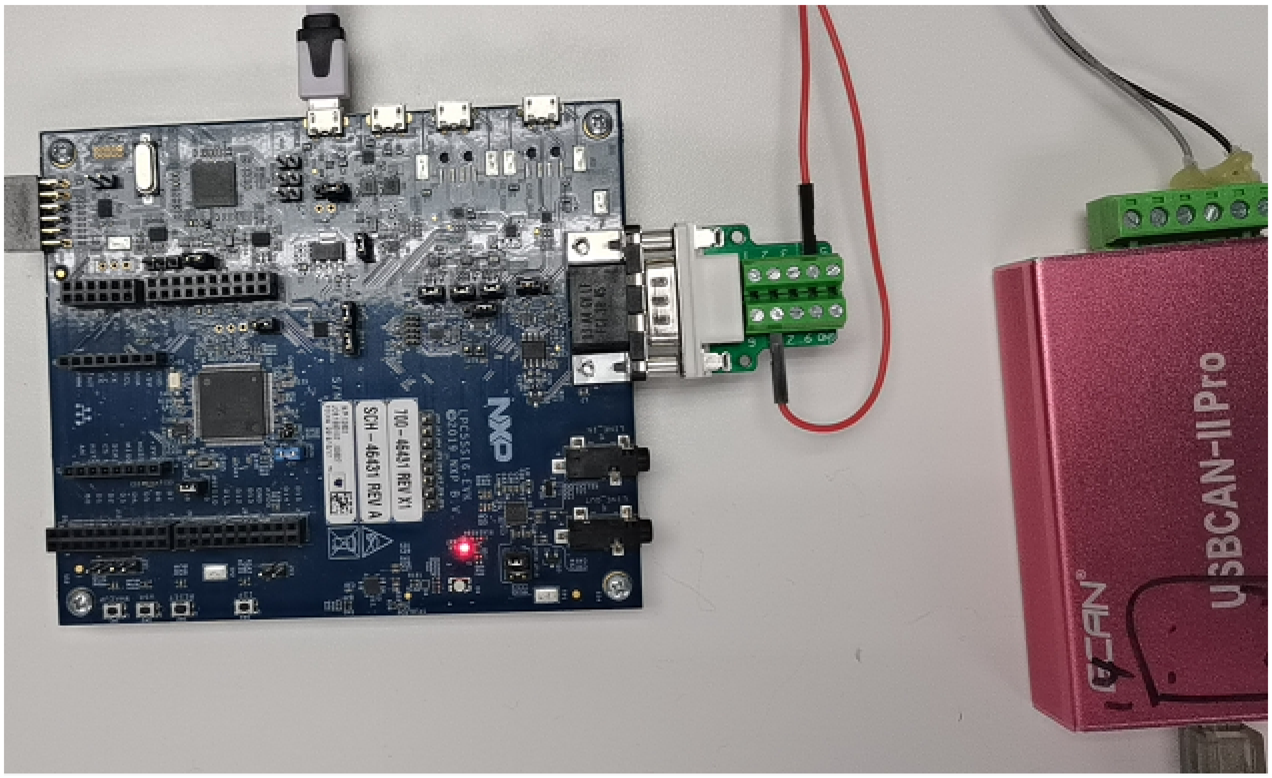
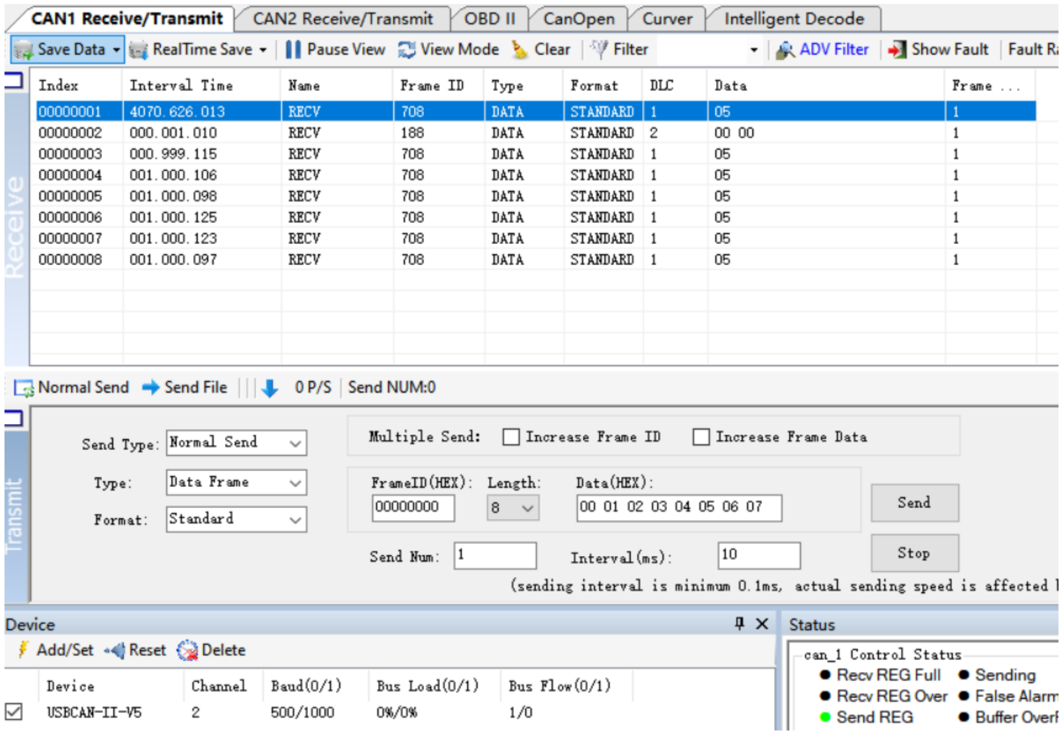


Figure 10. Message RAM configuration

3.3.2 Test Step

Download the program on to the board and press RESET button. Run MCU, open USB-CAN convertor GUI, you can see that the device is sending CANopen time sync frame every 1 s:



Index	Interval Time	Name	Frame ID	Type	Format	DLC	Data	Frame ...
00000001	4070.626.013	RECV	708	DATA	STANDARD	1	05	1
00000002	000.001.010	RECV	188	DATA	STANDARD	2	00 00	1
00000003	000.999.115	RECV	708	DATA	STANDARD	1	05	1
00000004	001.000.106	RECV	708	DATA	STANDARD	1	05	1
00000005	001.000.098	RECV	708	DATA	STANDARD	1	05	1
00000006	001.000.125	RECV	708	DATA	STANDARD	1	05	1
00000007	001.000.123	RECV	708	DATA	STANDARD	1	05	1
00000008	001.000.097	RECV	708	DATA	STANDARD	1	05	1

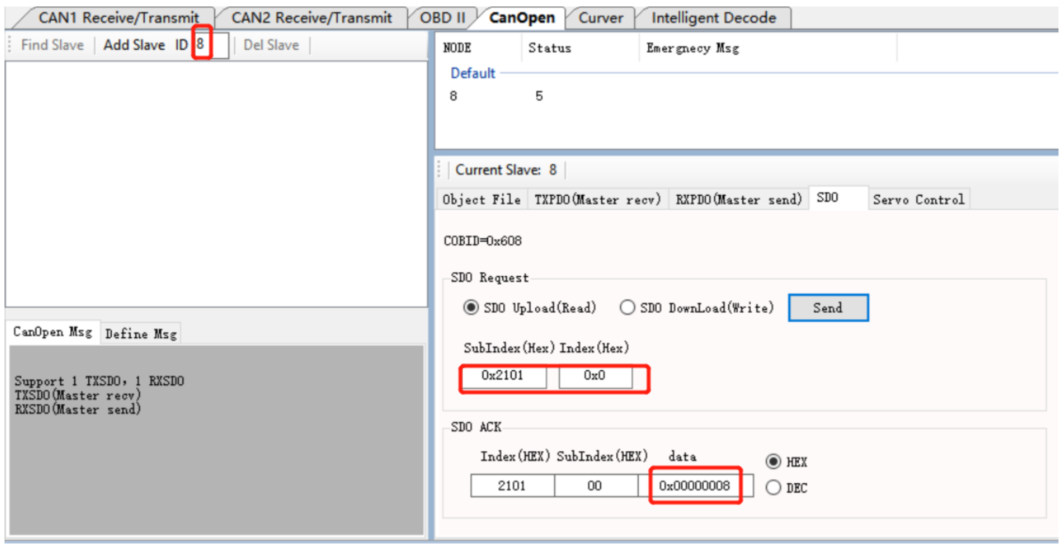
Send Type: Normal Send
Type: Data Frame
Format: Standard
Multiple Send: ☐ Increase Frame ID ☐ Increase Frame Data
FrameID(HEX): 00000000 Length: 8 Data(HEX): 00 01 02 03 04 05 06 07
Send Num: 1 Interval(ms): 10
(sending interval is minimum 0.1ms, actual sending speed is affected

	Device	Channel	Baud(0/1)	Bus Load(0/1)	Bus Flow(0/1)
☑	USBCAN-II-V5	2	500/1000	0%/0%	1/0

can_1 Control Status
● Recv REG Full ● Sending
● Recv REG Over ● False Alarm
🟢 Send REG ● Buffer Overf

Figure 11. Sync frame send from device every 1 s

Next, we test SDO functionality. Switch to CANopen panel, add Node 8, and click SDO panel. Set the operation type to SDO request, reading the value from OD 0x2101, and sub index is 0x00 (which store the CAN Node ID in OD). If everything OK, the return field should show 0x08.



Add Slave ID 8

NODE	Status	Emergnecy Msg
Default		
8	5	

Support 1 TXSDO, 1 RXSDO
TXSDO(Master recv)
RXSDO(Master send)

Current Slave: 8

COBID=0x608

SDO Request
[✓ SDO Upload(Read)] SDO DownLoad(Write)
SubIndex(Hex) Index(Hex)
0x2101 0x0

SDO ACK
Index(HEX) SubIndex(HEX) data
2101 00 0x00000008
[✓ HEX] DEC

Figure 12. SDO test

4 Reference

1. https://www.can-cia.org/canopen/
2. https://github.com/CANopenNode/CANopenNode

Date of release: 29 January 2021

Document identifier: AN13121