PAC25140 CANopen Manual

Power Application Controller®

2023-09-29

Contents

Contents

1. Introduction

CANopenNode is an Open-Source program written for 8-bit microcontrollers. It can be used in various devices connected on CAN bus (sensors, input/output units, command interfaces, various controllers etc.). Program is written according to CANopen standard, so devices can communicate with other devices based on CANopen.

1.1 Short description of CAN and CANopen

CAN (Controller Area Network) is serial bus system originally developed to be used in cars. It is also widely used in an industrial automation. Since it is cost effective (it is implemented inside many 8-bit microcontrollers) it can be used in wide range of applications.

Some features of CAN:

- cost effective,
- implemented in hardware,
- reliable (sophisticated error detection, erroneous messages are repeated, high immunity to electromagnetic interference),
- flexible,
- message length: max 8 bytes,
- messages have unique CAN identifier,
- arbitration without losing time, for example high priority message is send immediately after current message in transmission,
- data rate (cable length): 10kbps (5 km) to 1Mbps (25m),
- for connection of cables no hub or switch is needed. Devices can also be opto-isolated from network.

CAN reliability (statistic): If a network based on 250kbps operates for 2000 hours per year at an average bus load of 25% an undetected error occurs only once per 1000 years. [CANopenBook]

CAN is an implementation of lower layers of a communication. But when we need communication between devices, there are some issues: which identifiers to use, what are the contents of messages, how to handle errors, how to monitor other nodes, etc. To avoid *reinventing the wheel* there is CANopen.

CANopen is one of higher layer protocols based on CAN. It is Open, it means someone can use it and customize it as he wants. To comply the standard, minimum implementation is required. Anyway, CANopen offers many useful features, which can be used for good and reliable communication.

Some features of CANopen:

- With standard CAN identifier (11bit), up to 127 nodes can be on one network.
- It is not a typical master/slave protocol, so master is not necessary. However, some features can be used on one node only, for example SDO client. This node is usually used for configuration, and we can call it a master.
- **Object Dictionary (OD):** Inside OD are sorted variables, which are used by node and are accessible over CAN bus. OD has 16bit wide index and for each index 8bit wide subindex (for example variable `Device Type` has index 0x1000 and subindex 0x00). Variables can be accessed via CAN with **SDO (Service Data Objects)**. Variables can be read/write, read/only, etc. They can be retentive. Length of variables is up to 256 bytes (longer variables are transferred with segmented transfer). With standard CANopen Configuration tool, which runs on PC, OD is visible as tree, so device can be easy configured. With Object Dictionary a lot of variables can be accessed, but it is not the fastest way.
- **PDO (Process Data Objects)** are exchanged between nodes for fast communication. PDO is up to 8byte wide data object, transmitted from one node to that node, which are set to receive it. PDO can be send in different ways: periodically in time intervals, on change of state, synchronous with other nodes, etc. Node can transmit up to 512 PDOs and can receive up to 512 PDOs from other nodes (predefined connection set offers 4 TPDOs and 4 RPDOs).
- **NMT (Network Management):** Include, Boot-up message, Heartbeat protocol, and NMT message. Each node can be in one of four states: initialization, pre- operational, operational or stopped. For example, PDOs are working only in Operational state.
- **Error control – Heartbeat protocol:** It is for error control purposes and signals the presence of a node and its state. The Heartbeat message is a periodic message of the node to one or several other nodes. Other nodes can monitor if specific node is still working properly. (Besides, Heartbeat protocol there exists an old and out-dated error control services, which is called Node and Life Guarding protocol.)
- **Emergency message** is sent in case of error or warning in node.

For understanding CANopenNode further knowledge is required.
Microcontroller, C programming and CANopen protocol must be understood.

1.2 Features of CANopenNode

- **Macros for configuration:** Features can be configured with macros in CO_OD.h file. If feature is reduced or disabled, program and memory size is reduced.
- **CAN bit rates:** 10, 20, 50, 100, 125, 250, 500, 800, 1000 kbps; oscillator frequencies: 4, 8, 16, 20, 24, 32, 40, ... MHz; 8-bit Microcontroller can be stable also on 100% bus load on 1000 kbps bit rate.
- **CANopen conformance:** Comply to [CiADS301], [CiADR303-3]. Works with standard frame format (11bit identifier). Possible is RTR bit.
- **Service Data Objects (SDO):** SDO server and SDO client is implemented, expedited and segmented transfer. Variables can be long up to 256 bytes.
- **Object dictionary (OD):** It has two sides:
  1. CANopen side: Variables are accessed through index, sub-index and length via SDOs (read only, read/write etc.). Before written to memory, they can be verified for correct value.
  2. Program side: Variables have ordinary names (no index, sub-index). They can be in RAM Memory space, EEPROM or Flash Program space. EEPROM and Flash variables are retentive (keep value after power off) and can be changed via SDOs. Implementation is simple, fast and flexible.
- **Process Data Objects (PDO):** Implemented are TPDOs and RPDOs. Synchronous Transmission is automatic, other methods are possible. Parameters are COB-ID, Transmission Type, Inhibit time and Event timer. Mapping is static and must be 'hand made'. Length of PDO is calculated from Mapping.
- **SYNC object:** Producer or consumer with 1ms accuracy.
- **Network management (NMT):** Implemented, can operate with or without NMT master.
- **Heartbeat / Node guarding:** Heartbeat producer and consumer. (Monitoring of presence and NMT state of multiple nodes implemented). Node guarding is not implemented.
- **User CAN messages:** Freely usable RX or TX CAN messages.
- **Error control:** Emergency objects are used. Mechanism to catch different errors is implemented. For each error occurred, different flag bit is set and Emergency message is sent. According to flag bits, error register is calculated and device can be put in pre-operational. All Errors can be easily tracked through Object Dictionary Entry (index 0x2100). This mechanism can also be used for user defined errors.
- **Status LED diodes:** Green and Red diodes according to [CiADR303-3].
- **Example for Generic Input/Output device:** Digital I/O, Analog I/O, Change of state transmission with event and inhibit timer. According to device profile CiADS401.

2. Design

CANopenNode is currently applied for PAC25140 BMS.

Implemented is CANopen and frame for user program. Goal is to be simple, powerful and open for extensions.

CANopenNode is Open Source. License used is LGPL, that means license acts only on library, not on complete user program.

2.1 Files

- **…\pac25xxx\CANopenNode (folder)** – CANopenNode source code. It includes the documents and source files for CANopen with example:
  - **doc (subfolders)** – documentation:

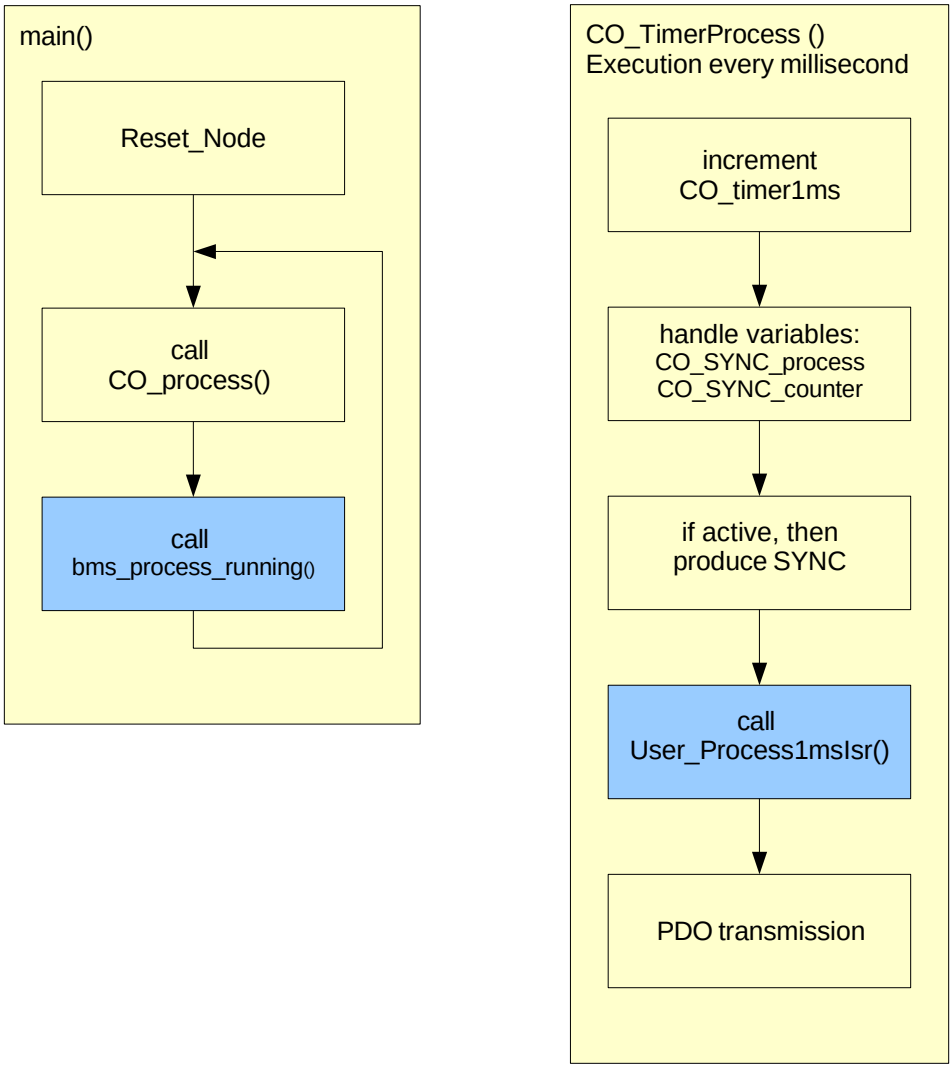
    CANopenNode Manual.pdf – This manual in pdf format.
  - **stack (subfolders)** – includes code files for majority protocols of CANopenNode:
    - **Emergency**
    - **HBconsumer**
    - **Heartbeat**
    - **PDO**
    - **SDO**
    - **SYNC**
  - **Other files** – license and source files:
    - **CANopen.h** – main CANopen header file. Included are general definitions and other headers.
    - **CANopen.c** – main CANopen processor and configuration.
    - **CO_driver.h** – Processor / compiler specific macros.
    - **CO_driver.c** – Processor / compiler specific functions.
    - **CO_OD.h** – header for Object Dictionary and main setup for CANopenNode.
    - **CO_OD.c** – variables, verify function and Object Dictionary for CANopenNode.
  - **LICENSE** – GNU GENERAL PUBLIC LICENSE file.

2.2 Program flow chart (mainline and timer procedure)

CANopenNode program execution is divided into three(four) tasks:

1. After startup, basic task is mainline. It is executed inside endless loop. In `CO_process` function not-time-critical program code is processed. All code is non-blocking.
2. Second task is Timer procedure which is triggered by interrupt and is executed every 1 millisecond. Here is processed time-critical code. Code must be fast. If execution time is longer than 1ms, overflow occurs, error bit is set and Emergency message is sent.
3. Third and fourth tasks are CAN transmit/receive interrupts. CAN receive interrupt must have priority over timer procedure.

Fields with blue background are functions, where user code can be written. Besides, that user can define other interrupts (and in multitasking systems other tasks, of course).

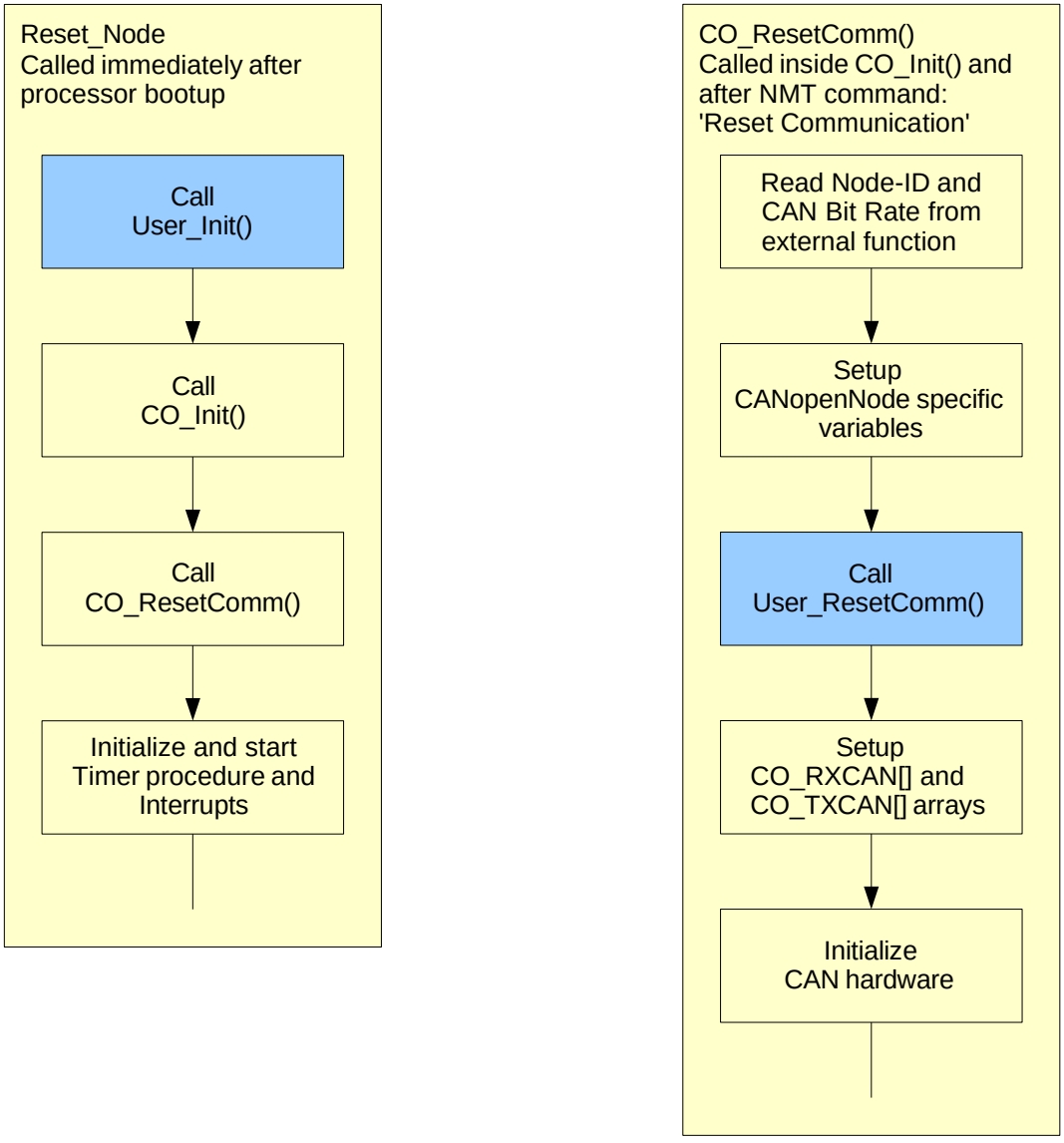


Picture 2.1 – Flow chart for mainline and timer procedure

2.3 Reset Node and Reset Communication procedures

In CANopen two different resets are defined: 'Reset Node' and 'Reset Communication'. Both can be triggered with NMT command from NMT master node. First reset is complete processor reset and is executed also after processor bootup, second is partial reset and is used for communication reset. In `CO_ResetComm()` all CANopenNode specific variables are setup. So, for example if PDO communication parameters has been changed, 'Reset Communication' must be performed for changes to take effect.

Fields with blue background are functions, where user code can be written.



Picture 2.2 – Reset Node and Reset Communication procedures

2.4 CAN messages, receiving/transmitting

Receiving and transmitting of CAN messages is made with interrupt functions. These functions are hardware specific. They have integrated some CANopen specific code.

2.4.1 Variables for CAN messages

Main variables for CAN messages are `CO_CANmodule_rxArray0[]` & `CO_CANmodule_txArray0[]` arrays (rx = receive, tx = transmit). Number of members in each array is equal to number of different CANopen communication objects (COB=communication object). Everything in CANopenNode 'turns around' these two arrays. For description, which COBs are used in each array, see CAN_driver.h file.

```
static CO_CANrx_t          *CO_CANmodule_rxArray0;
static CO_CANtx_t          *CO_CANmodule_txArray0;
```

Type of one element in each array is described below:

```
typedef struct{
      union{
            uint32_t RXBUF0;
            struct{
                  unsigned    DLC   :4;
                  unsigned          :2;
                  unsigned    RTR   :1;
                  unsigned    FF    :1;
                  unsigned    ident :16;
                  unsigned    data0 :8;
            };
      };
      uint8_t     data[8];
}CO_CANrxMsg_t;


typedef struct{
      union{
            uint32_t        TXBUF0;
            struct{
                  unsigned    DLC   :4;
                  unsigned          :2;
                  unsigned    RTR   :1;
                  unsigned    FF    :1;
                  unsigned    ident :16;
                  unsigned    data0 :8;
            };
      };
      [...]
      uint8_t         data[8];
      volatile bool_t bufferFull;
      volatile bool_t syncFlag;
}CO_CANtx_t;
```

Besides, there are 2 communication flags are declared in CO_driver.h:

```
typedef struct{
      [...]

      volatile bool_t     bufferInhibitFlag;

      volatile bool_t     firstCANtxMessage;

}CO_CANmodule_t;
```

`ident` is CAN message standard identifier aligned with hardware registers. It includes 11 bit COB-ID and Remote Transfer Request bit (RTR). For alignment use macros `CO_CANinterrupt()` from CO_driver.c. Message is received if CAN-ID and RTR are matched.

`DLC` is length of data in message (CAN uses 0 to 8 bytes). For receiving there is a special rule: if value is greater than 8, length of message is not checked and thus message with any length is accepted.

`firstCANtxMessage` is a flag: for reception this bit is set when new message has received, for transmission this bit 'informs' tx interrupt procedure, that this message is ready to be sent.

`firstCANtxMessage` flag has different meaning. For reception, data from new message received from bus are not copied if (`bufferInhibitFlag==1` AND `firstCANtxMessage==1`). For example, if old message was not read yet, new message is lost. For transmission, flag is used for synchronous TPDO messages. Message is destroyed and is not send over CAN, if `bufferInhibitFlag`==1 AND time is outside SYNC window (OD, index 1007h).

`data` is 8 bytes of CAN data.

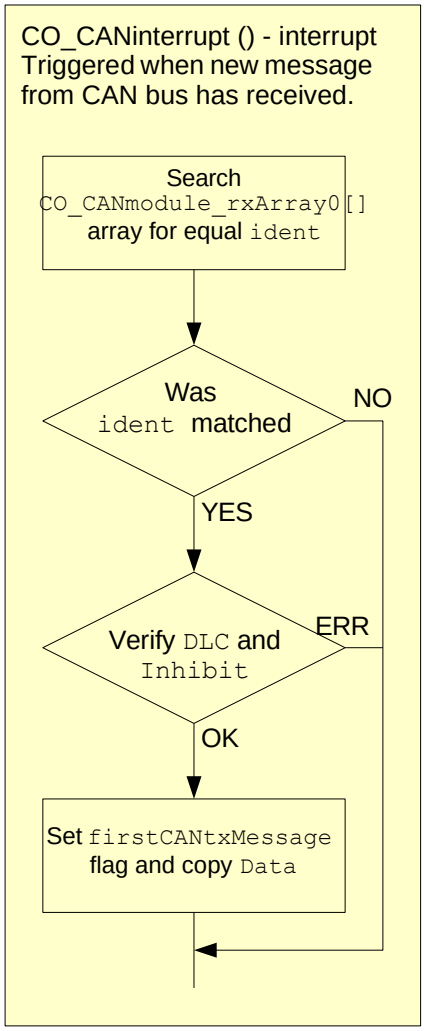
At communication reset first whole array is cleared and then all values except Data are initialized (inside `CO_ResetComm()` function). This way minimal data copying is achieved, and no other buffers are needed.

2.4.2 Reception of CAN messages

CANopen uses many different CAN identifiers, so hardware filtering of messages cannot always be used. Instead, filtering is implemented in software. Principle is the following: Identifiers for all COBs, used with current configuration, are written in `CO_CANmodule_rxArray0[]` array at startup. When new message arrives from CAN bus, `CO_CANinterrupt()` interrupt is triggered and array is searched from first to last element. If both COB-IDs, from message and array, are matched, message goes into further processing. If not, interrupt exits.

This interrupt must be high priority and must have low latency, because many messages must be filtered out. In CANopenNode it is implemented with fast assembly code, so there is no problem even with high CAN bit rates and high bus load.

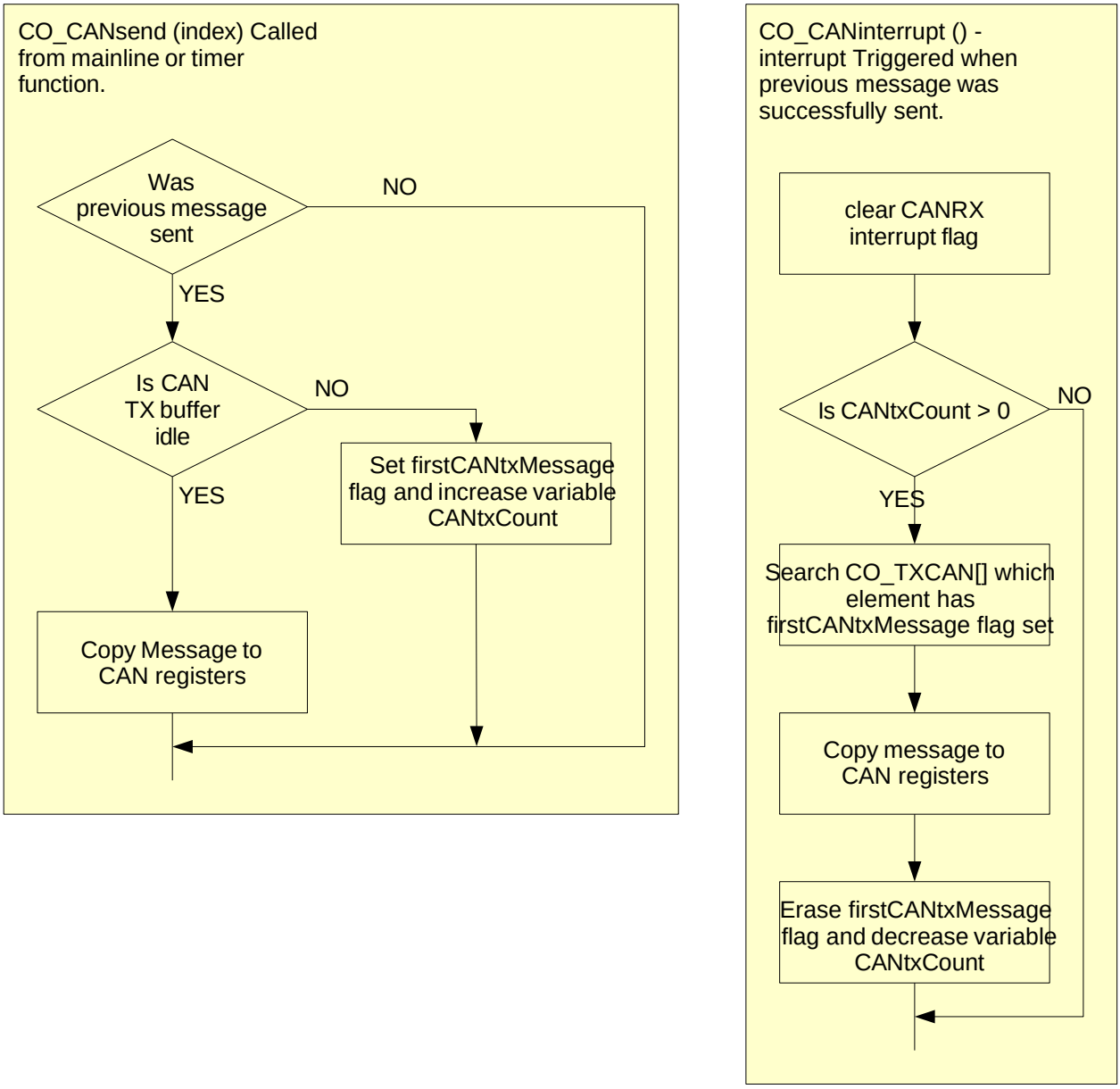
Received messages are then handled inside different functions. Details will be described in next chapters.



Picture 2.3 – Reception of CAN messages

2.4.3 Transmission of CAN messages

Function, which wants to send CAN message, first prepares adequate `CO_CANmodule_txArray0[]` array element: writes Data and, if necessary, it can also modify other parameters. Then it calls `CO_CANsend()` function. If CAN TX buffer is free, message will be sent immediately, otherwise it will be marked, and interrupt will send it. Messages with lower `index` (`CO_CANmodule_txArray0[index]`) will be send first.



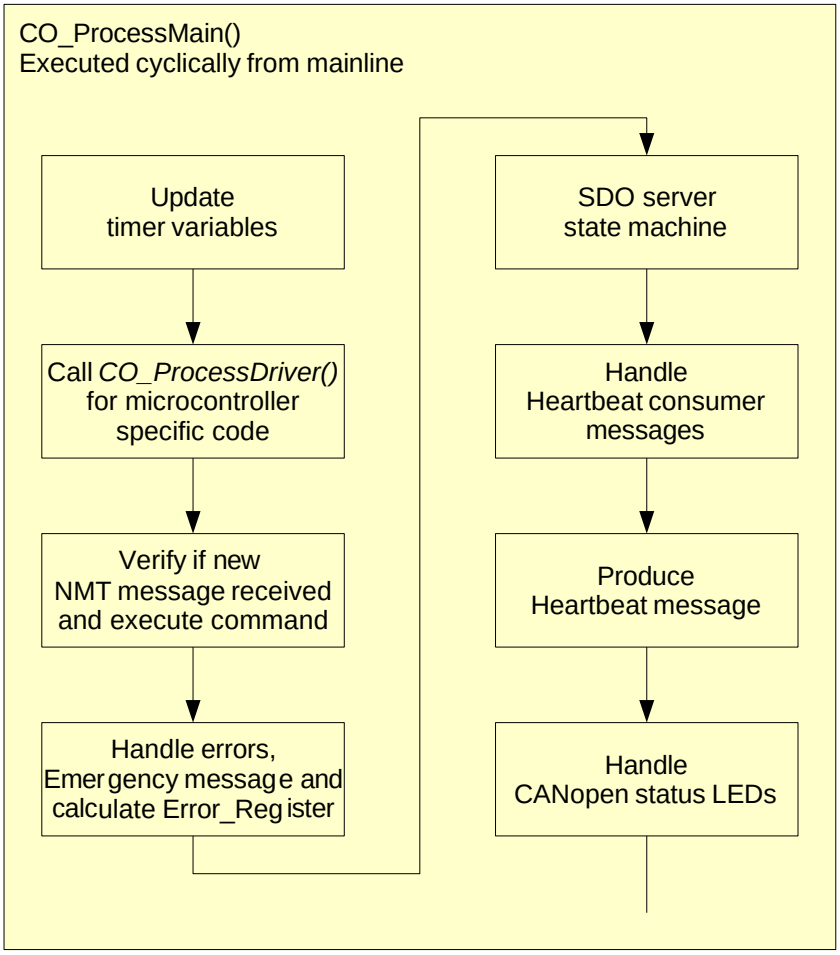
Picture 2.4 – Transmission of CAN messages

2.5 Mainline procedure

As mentioned in chapter 2.2, there are two 'tasks' handling CANopen messages. Timer procedure is shorter and is described in that chapter, mainline procedure is described here.

`CO_process()` is executed inside endless loop. There is processed program code, which is not time-critical. All code is non-blocking. Sometimes a lot of code has to be processed, so this can lead to longer delays and longer execution cycle. For example, SDO communication is very time consuming.

In same loop also user function `bms_process_running()` is processing. There can also be time consuming and not-time-critical code. Anyway, blocking functions must **not** be used.



Picture 2.5 – Flow chart of CANopenNode mainline procedure

2.6 COB – Communication Objects

Pre-defined Connection Set in CANopen connects Communication Objects with their identifiers. It applies to the standard CAN frame with 11-bit Identifier. Especially for PDOs it is not the rule to use Pre-defined values.

Object	COB-ID
NMT SERVICE	000h
SYNC	080h
EMERGENCY	080h + NODE ID
TIME STAMP	100h
TPDO1	180h + NODE ID
RPDO1	200h + NODE ID
TPDO2	280h + NODE ID
RPDO2	300h + NODE ID
TPDO3	380h + NODE ID
RPDO3	400h + NODE ID
TPDO4	480h + NODE ID
RPDO4	500h + NODE ID
TSDO	580h + NODE ID
RSDO	600h + NODE ID
HEARTBEAT	700h + NODE ID

Table 2.1 – Pre-defined Connection Set

2.6.1 NMT and network management

Network management is detailed in CANopen standard. In general, each node has 4 possible states: Initialization, Pre-Operational, Operational and Stopped. In Operational state, all communication is allowed. In Pre-Operational state, all communication is allowed, except PDOs. In Stopped state, only NMT messages can be received and processed. Handling those states is quite tricky, especially if reliable communication is required.

By default, node enters Operational state after bootup. This is set with variable 'NMT startup' in Object Dictionary at index 0x1F80. If bit 2 is set, node will start Pre-Operational after bootup.

There is another issue. If there is an error in the node, node sometimes should not enter into Operational. This happens, when Error Register (OD, index 0x1001) is set. More about that is in 'EMERGENCY' chapter.

In CANopenNode received NMT messages are handled in `CO_Process()` function. According to command, actions are triggered. NMT master can be easily implemented with user defined CANTX message. NMT master can send specific NMT command to single node or to all nodes (broadcast).

2.6.2 SYNC

SYNC message is useful for synchronization. One node on the network sends it periodically in constant time intervals. It can be used for different purposes, one of them is using Synchronous PDOs.

SYNC message is integrated into CANopenNode. It is handled inside Timer procedure. Basic time unit is 1 millisecond. CANopenNode can be a producer or consumer. There are two useful variables related to SYNC: `CO_SYNC_process` and `CO_SYNCcounter` (type unsigned int). `CO_SYNC_process` is incremented each millisecond and reset to zero, each time SYNC is received/transmitted. `CO_SYNCcounter` is incremented each time, SYNC is received/transmitted.

2.6.3 EMERGENCY and error handling

Error handling is very useful thing in practical implementation of network. In testing phase there can be some bugs in software and even some errors in electronic or mechanic parts of network. With error handling many of those errors can be tracked.

In CANopenNode error handling is made simple. In general when error occurs in a node, node should not crash, it should operate further.

Principle is the following: if somewhere in program occurs error (certain condition is met), `CO_errorReport()` function is called, which just sets some variables. When in turn, error handling is processed inside `CO_Process()` function. All Error handling specific definitions are collected in file `CO_Emergency.h`.

Following function accepts two parameters:

```
void CO_errorReport(CO_EM_t *em, const uint8_t errorBit, const uint16_t
errorCode, const uint32_t infoCode);
```

`errorBit` is unique for each different error situation, `Code` is customer specific additional information about the error. As said before, this function just sets some variables, besides others it sets appropriate bit in `errorStatusBits[]` array. If that bit was already set before, nothing happens. This way specific error is reported only the first time, later repeating is ignored. Opposite function to `CO_errorReport()` is `CO_errorReset()`, where specific bit is cleared if error is solved somehow.

If new error occurred, procedure inside `CO_Process()` function verifies new situation. First, Error Register (OD, index 0x1001) is calculated from `errorStatusBits[]` array. Then EMERGENCY message is sent. Emergency is also written to history (OD, index 1003).

EMERGENCY message is 8 bytes long. First three bytes are standard: First two bytes are Emergency Error Code, Third byte is Error Register. 4th byte is `ErrorBit` and 5th-6th bytes are Code from `CO_errorReport()` function. 7th and 8th byte can be user specific.

Each bit in 1-byte long Error Register is calculated from state of `errorStatusBits[]` array. See `CO_errorRegisterBitmask_t` macros in `CO_Emergency.h` file. When Condition for specific bit is met, that bit is set and vice versa. In some cases, those conditions can be more restrictive, in other cases not.

If Error Register is not equal to zero, node is prevented to be in Operational state, and PDOs cannot be transmitted or received.

If macro `OD_errorStatusBits` in `CO_OD.h` file is set to 1, Communication error bit in Error register will not prevent node to be in Operational state. In this case node will always stay operational, even

if disconnected from network.

2.6.4 TIME STAMP

Not implemented in CANopenNode.

2.6.5 PDO – Process Data Objects

Process Data Objects are optimized for frequent transmission of Data over the network. In comparison with Service Data Objects they are faster, shorter, no protocol overhead and need no answer. Whole 8-bit wide CAN data field is used for Process Data.

CANopen uses two parameters for setup PDOs: PDO Communication Parameters and PDO Mapping Parameters. For proper using of PDOs understanding of them is required. They are described in other Literature.

Mapping parameters describes which data from Object Dictionary are used for specific PDO. In CANopenNode all mapping is static, this means that mapping cannot be changed, after program is build. In CANopenNode mapping parameters are used for calculating length of PDOs. When RPDOs are received, length must match to that specified in mapping or RPDO will not be processed and emergency will be sent first time. If mapping parameters are disabled by disabling macro `RPDOMapPar`, length of TPDOs will be fixed to 8 bytes and any length of RPDO will be accepted. Mapping parameters can also be disabled in CO_OD.h file. In this case, TPDO length will be fixed to 8 bytes and any RPDO length will be accepted.

Communication Parameters describes CAN ID for PDO communication object (COB-ID), Transmission Type, Inhibit time and Event timer.

COB-ID on transmitting node must match COB-IDs on all receiving nodes. On CANopen is only one rule for COB-IDs: There must not exist two equal COB-IDs on one network for transmitted messages. Anyway, CANopen has pre-defined connection set, where for 4 TPDOs and 4 RPDOs COB-IDs are defined. In CANopenNode standard or custom values can be used. If bit 31==1, PDO is not used.

Transmission Type describes, when PDO will be transmitted. There are more possibilities: Time intervals, Change-of-state, combination of them, after Remote transmission request (rtr – CAN feature), or synchronous after SYNC message.

PDO transmission in CANopenNode

Many options are possible.

Synchronous transmission is implemented in CANopenNode. Messages are transmitted automatically after every n-th SYNC message (n = value of Transmission Type variable 1 ... 240). It is predictable.

Event timer sends message in time intervals and is automatic in CANopenNode.

One possibility is combination of *Change-of-state* and (longer) Time intervals. This is good in cases, where changes are not very often. Advantage is fast response and low traffic. Problems are for example with values from analog sensors, where least significant bit changes often. This leads to many unnecessary messages. Another problem is that bus load cannot be always predicted. This method is used in Example with generic I/O.

Data, which are sent with TPDO, must be written manually before PDO is sent. They must be written in array `CO->TPDO[i]`, where index is PDO number (0 is first PDO).

To send PDO manually, just call the following function:

```
int16_t CO_TPDOsend(CO_TPDO_t *TPDO);
```

where `*TPDO` is PDO number (0 is first PDO). If return == 0, transmission was successful.

PDOs are (and may be) transmitted only when node is in Operational state.

PDO reception in CANopenNode

When PDO arrives from CAN bus, it is memorized. Anyway, PDOs may only be used when node is in Operational state.

RPDO data can be read from `CO->RPDO[i]` array, where index is PDO number (0 is first PDO).

`CO_RPDO_New(i)` array element is set to 1 every time PDO is received. User may manually erase it.

User must pay attention on one issue: PDO is received with high priority interrupt, so if user reads data and interrupt occurs during read, data can be unpredictable. So, during read, CANrx interrupt should be disabled (see `CO_driver.h`).

Another issue is rule in CANopen standard, where synchronous PDOs must be processed after next SYNC message. But after next SYNC message also next RPDO can arrive and overwrite old PDO. Solution is: inside Timer1ms procedure save all new RPDOs to different location except if `CO_SYNC_process == 0`. If so, process previously copied RPDOs.

There is no control, if node is in operational state, received PDO is saved always.

To make sure PDO reception is working correctly, following conditions must be met before read of RPDO Data: Both, this node and transmitting node, must be in operational state. Checking of `CO_RPDO_New(i)` is thus not necessary. For scanning other nodes use Heartbeat consumer.

2.6.6 SDO – Service Data Objects

With SDO whole Object Dictionary of any node on the CANopen network can be accessed. SDO client (master) have access to SDO servers on other nodes.

SDO server is integrated in CANopenNode. Maximum variable length can be set from 4 ... 256 bytes. If max. length is 4 bytes, then only expedited transfer is used. Otherwise, segmented transfer is used and more flash memory is consumed. More than 1 SDO channel can be used.

Also, SDO client non-blocking functions are available, so CANopenNode can be used as master too. For example how to use them, see functions `CO_SDOclientDownload ()` and `CO_SDOclientUpload()`.

SDO communication have access to many variables, but it is not very fast. It is quite time consuming especially in PIC microcontrollers. But it is very powerful for setup the node.

Design in CANopenNode: SDO server is a state machine implemented inside mainline function. When new SDO object arrives from client, it is processed and some static variables are set. According to command and state it: searches Object Dictionary (OD) for entry with correct index and subindex, make verifications, send aborts if necessary, collect data segments, reads or writes to variables from OD, etc. For CANopenNode user no detailed knowledge about SDO objects is needed.

2.6.7 Heartbeat

Heartbeat protocol is used for monitoring proper operation of remote nodes. In CANopenNode are implemented Heartbeat producer and consumer. Old-dated Node Guarding is not implemented.

With Heartbeat no master is needed and each node can monitor nodes, which are important for it.

To setup Producer, corresponding entry in OD must be edited. Node will produce periodic Heartbeat messages.

Heartbeat consumer monitors presence and state of nodes defined in corresponding entry in OD. Monitoring starts after reception of the first Heartbeat from specified node. If Heartbeat does not arrive in specified time, emergency message is generated. Heartbeat consumer time value should be set to 1,5 * Heartbeat producer time value on remote node. State of remote node can be read from `CO_HBcons->NMTstate(i)` array.

2.7 Object Dictionary

One of the most powerful features of CANopen is Object Dictionary. It is like a table, where are collected all network-accessible data. Each entry has 16bit index and 8bit subindex. Data type of each entry can be different - from 8 bit unsigned to string.

In CANopenNode Object Dictionary is an array of entries `CO_OD[]`. Each entry has an information about index, sub-index, attribute (read only, read/write etc.), length and pointer to data variable. There are “Two sides” of Object Dictionary:

1. Program side: Variables have ordinary names (no index, subindex) and types.
2. CANopen side: Ordinary variables are collected in `CO_OD[]` array and so SDO server have access to them through index, sub-index and length (read only, read/write etc.).

2.7.1 Memory types of variables

In CANopenNode three types of variables are used in Object Dictionary:

- Variables allocated in program memory flash space (ROM variables).
- Variables allocated in RAM space.
- Variables allocated in RAM space and saved-to/loaded-from EEPROM.

1. Features of ROM variables:
    - Keep value after power off.
    - Initialization values are written when chip is being programmed.
    - Readable as usual variables.
    - Writable in run time with SDO objects.
    - Writing to variable by user program is not possible directly.
    - Possible problem: if device resets during write data can be corrupted.

2. Features of RAM variables:
    - Classic read/write.
    - After power off value is lost.
3. Features of EEPROM variables:
    - Variables are read from (written to) RAM (allocated inside CO_OD_EEPROM structure).
    - At chip initialization values are read from EEPROM.
    - At run time background routine compares RAM and EEPROM and saves byte by byte.
    - Classic writing to variable by user program.
    - Possible problem: if microcontroller resets during write: multi byte variable can be corrupted.

Reading and writing to different types of variables is processor specific, so these two functions are different for different microcontroller. They are placed in `CO_driver.c` file.

2.7.2 Connection between variables and Object Dictionary

- To each variable is assigned one entry of `CO_OD[]` array;
- Entry holds information about index, subindex, length and pointer to variable.
- When SDO server wants to find entry (with specific index and sub-index) inside object dictionary, it calls `const CO_OD_entryRecord_t` and function returns pointer to that entry.

2.7.3 Verify function

Function `VerifyODwrite()` is called from SDO server, when a write to Object Dictionary entry is in progress. Function is called when data, that came from network are known. Function verifies if data value is correct and then returns SDO Abort Code. If returns 0, data are correct and are written to Object Dictionary. Function does not verify length - it is verified by SDO server.

3. Compiler and Hardware connection

Main hardware specific files are:

- **main.c** – initialization and definition for interrupt, timer1ms and main functions.
- **CO_driver.h** – Processor / compiler specific macros.
- **CO_driver.c** – Processor / compiler specific functions.

These files contain:

- Base for main and interrupt functions.
- Different software and interrupt enable/disable macros.
- Function for reading Node-ID and Bitrate from hardware.
- Function for initialization of CAN controller.
- Function for writing message to CAN controller.
- Interrupt for receiving messages from CAN controller and identifying them. It must be fast.
- Handling errors from CAN controller.
- Read/write data from/to Object Dictionary. Data can be in RAM or in ROM memory.
- Handling EEPROM variables.

3.1 Compiler

The PAC25140 contains an Arm Cortex-M4F. CANopenNode is applied for it with integrated CAN module.

Compiler used is GNU Arm on Eclipse IDE v4.27.

Memory types used are RAM, ROM and EEPROM. Read/write via SDO is fully supported.

ROM variables uses program flash memory and utilizes self-programming feature.

EEPROM variables are on startup read from internal EEPROM to RAM.
If RAM is changed EEPROM is automatically updated.

3.2 Hardware connection

PLL setting: **50MHz**
CAN baud rate setting: **125kbps.**

PAC25140 device connect to CANalyst-II module via CAN transceiver, connection as below:

PAC25140	**CAN transceiver**	**CANalyst-II**
VCC33 (3.3V)	3V3	-
GND	GND	-
GPIOE.P2	RXD	-
GPIOE.P3	TXD	-
-	CANH	CAN1H
-	CANL	CAN1L

4. Examples and Object Dictionary

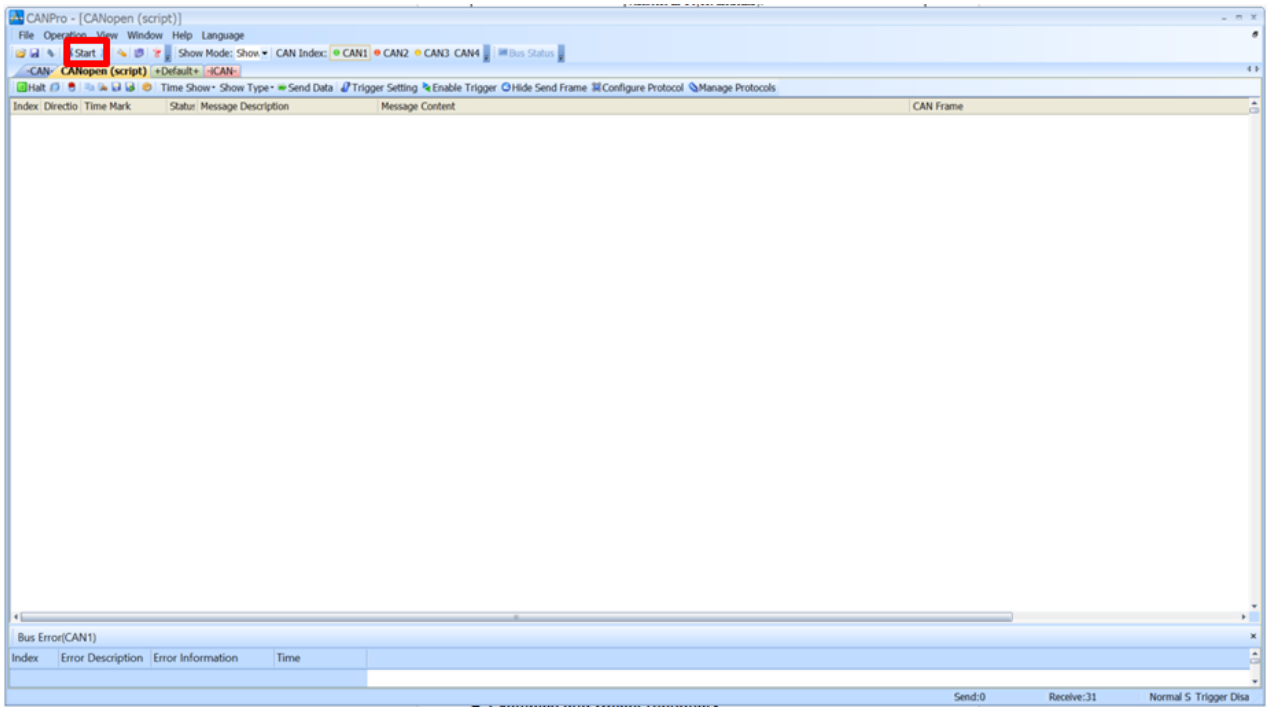
4.1 NMT Module Control protocol

NMT Message		
CAN ID	Data	
	Byte 0	Byte 1
000	<CMD>	<NodeID>

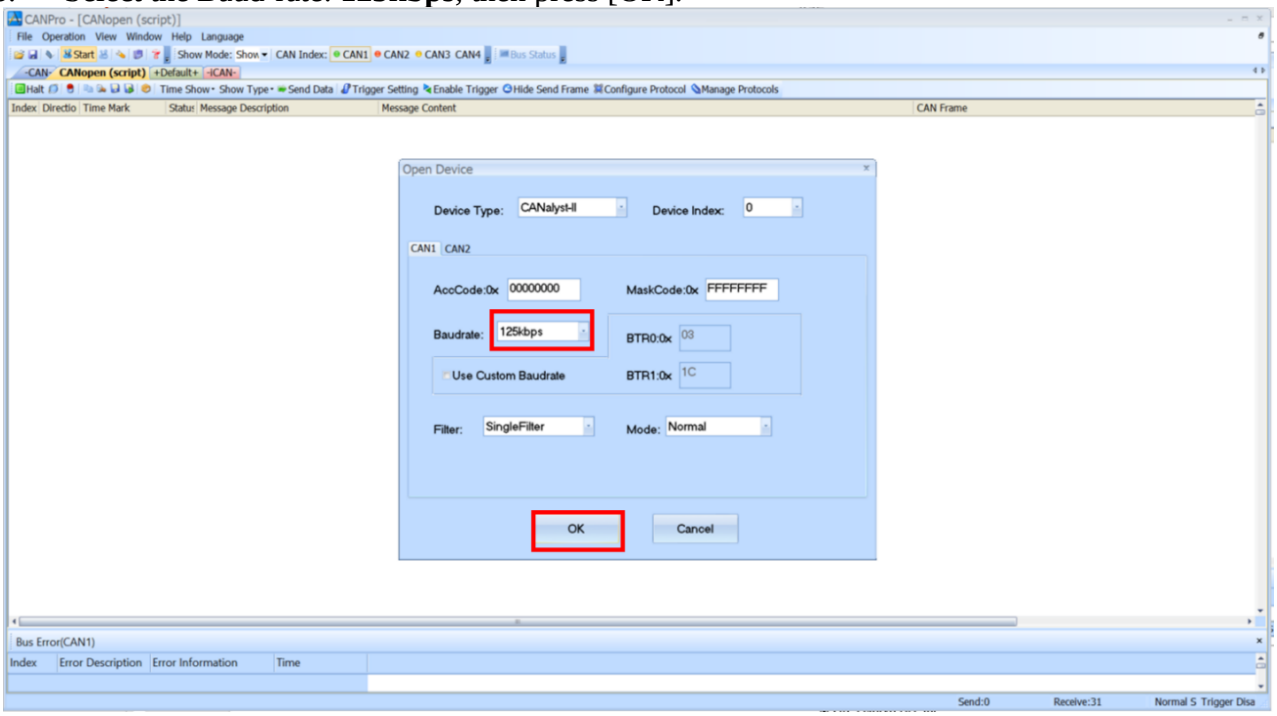
<CMD>	Meaning
01 h	Switch to the "Operational" state
02 h	Switch to the "Stop" state
80 h	Switch to the "Pre-Operational" state
81 h	Reset Node
82 h	Reset Communication

4.1.1 Start Remote Node

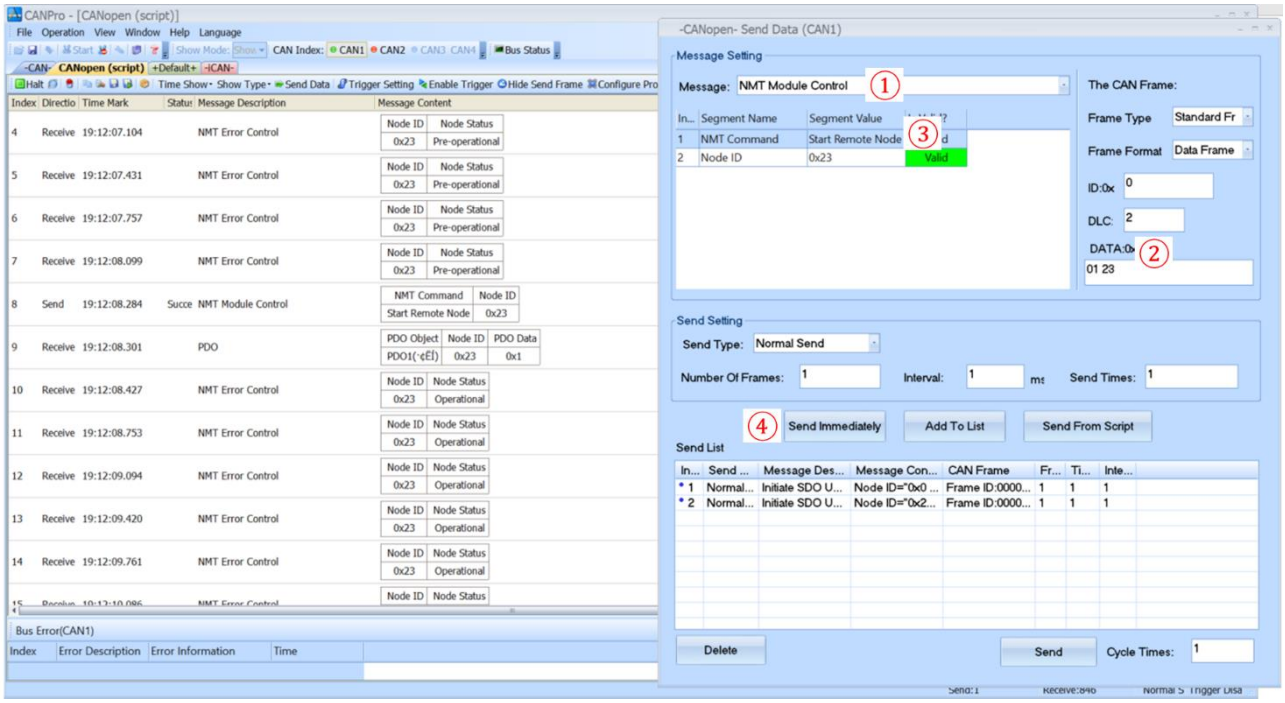
1. After connecting PAC25140 to CANalyst-II, open CANPro Analyzer.
2. Select [Start] on the taskbar.



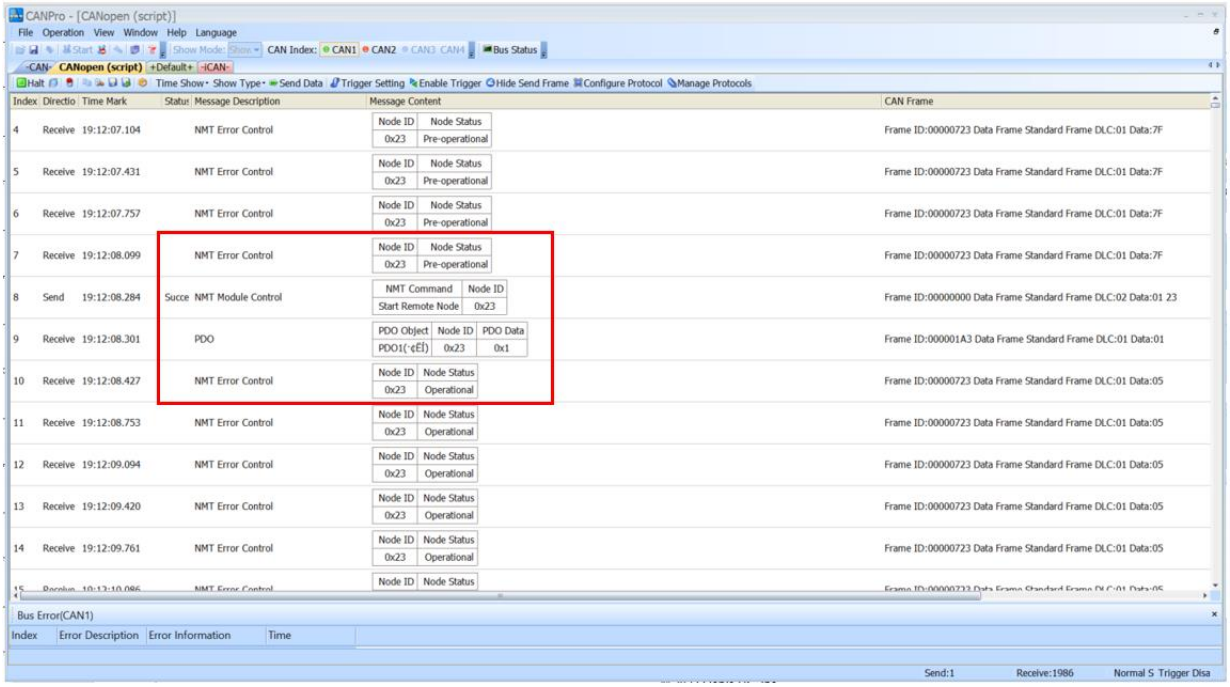
3. Select the Baud-rate: **125kbps**, then press [OK].



4. Select [Send Data] → 4.1. In the send data window Select Message [NMT Module Control] → 4.2. Set DLC:2 → 4.3. Select NMT command [Start Remote Node], Note ID: 0x23→ 4.4. Send Immediately.



5. Node Status is change from [Pre-Operational] to [Operational].

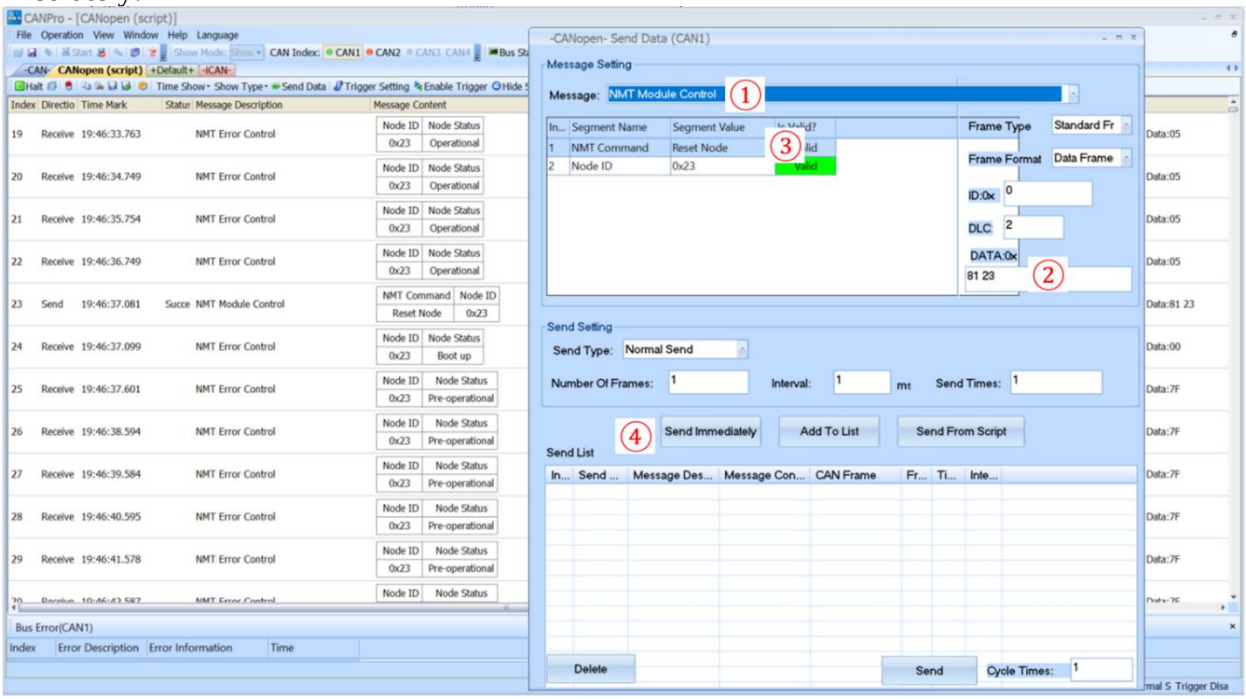


6. Protocol:
   - Frame ID: 00000000
     → CAN ID: 000
   - Data Frame: 01 23
     → CMD: Start Remote, Node ID: 0x23

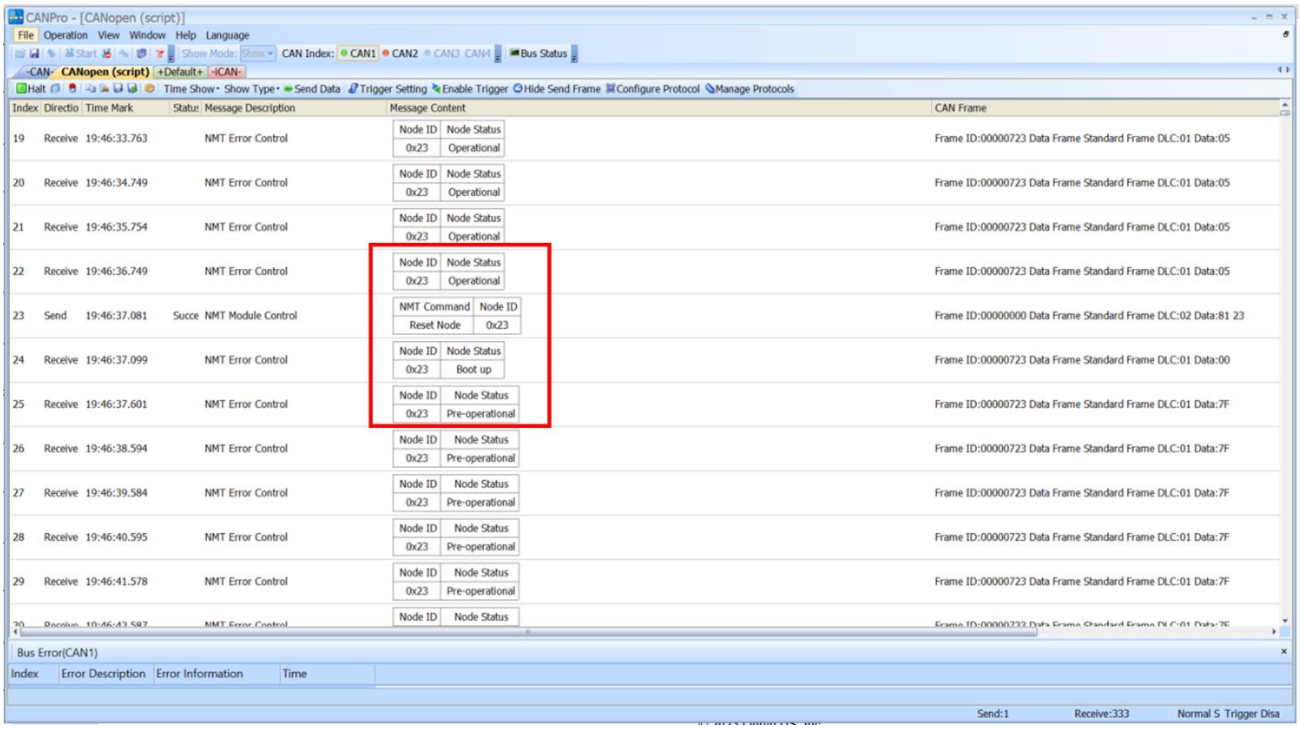
4.1.2 Reset Node

Similar with *Start Remote Control* for step 1-3.

4. Select [Send Data] → 4.1. In the send data window Select Message [NMT Module Control] → 4.2. Set DLC:2 → 4.3. Select NMT command [Start Remote Node] - Note ID: 0x23 → 4.4. Send Immediately.



5. Node Status is change from [Operational] to [Boot-up], then change to [Pre-Operational].



6. Protocol:
   - Frame ID: 00000000
     → CAN ID: 000

   - Data Frame: 81 23
     → CMD: Reset Node, Node ID: 0x23

Similar for **Stop Remote Node**, **Pre-operational state** and **Reset Communication**.

4.2. Initiate SDO Upload/Download

4.2.1. Initiate SDO Download

Client Server

Init SDO Download request

CAN ID	Data							
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
600+NodeID	<CMD>	<IDX>		<SUBIDX>	<Data>			

Here, the <CMD> byte is dependent on the length of the data that are to be written. <CMD> can be one of the following values:

- 1 byte data length: 2F h
- 2 byte data length: 2B h
- 3 byte data length: 27 h
- 4 byte data length: 23 h

Client Server

Init SDO Download confirm (OK)

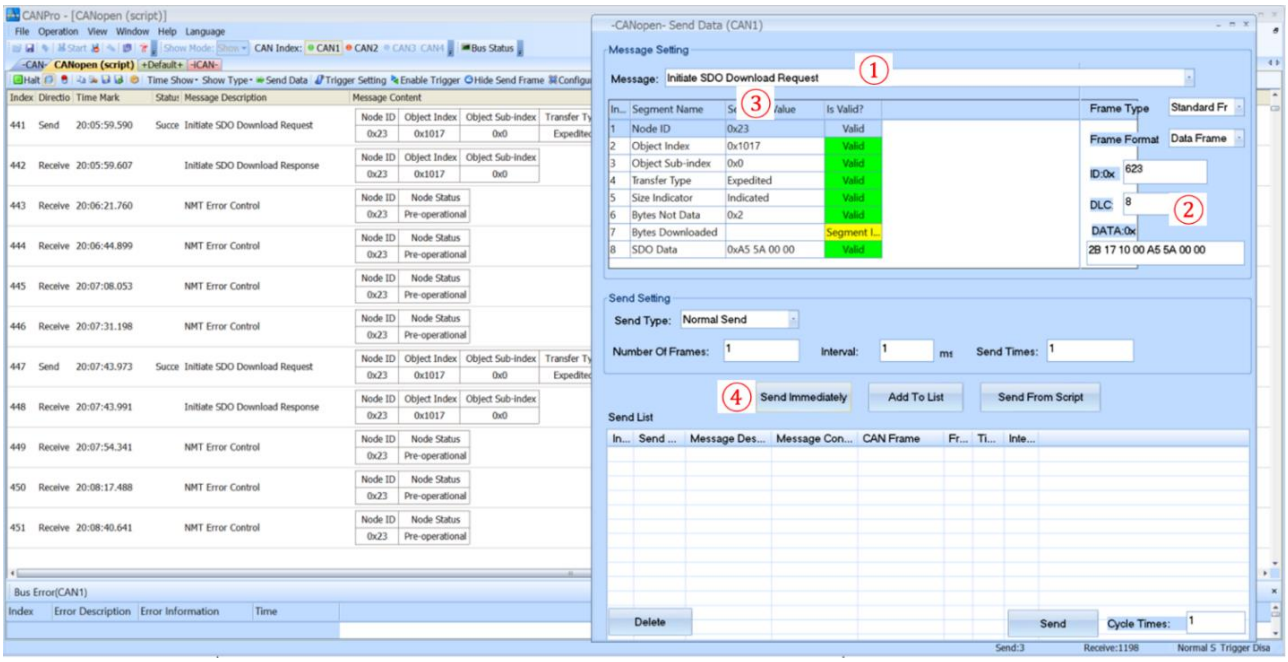
CAN ID	Data							
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
580+NodeID	0x60	<IDX>		<SUBIDX>	0x00	0x00	0x00	0x00

Init SDO Download confirm (error)

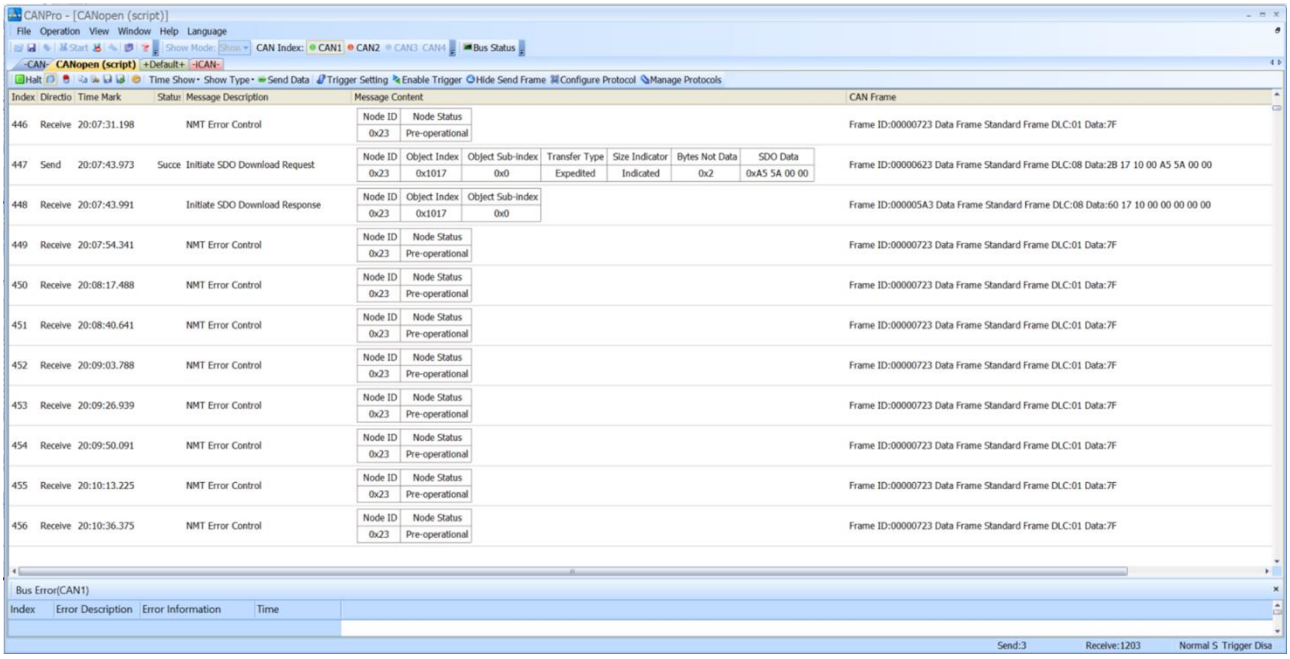
CAN ID	Data							
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
580+NodeID	0x80	<IDX>		<SUBIDX>	<ERROR CODE>			

Similar with *Start Remote Control* for step 1-3.

4. Select [Send Data] → 4.1. In the send window Select Message [Initiate SDO Download Request] → 4.2. Set DLC: 8 → 4.3. Set Note ID: 0x23 - Index: 0x1017 (2bytes) - Sub-index: 0 - Transfer type: Expedited - Size indication: Indicated - Bytes Note Data: 0x02 - SDO Data: 0xA5 5A → 4.4. Send Immediately.



5. CANalyst send a request download to PAC25140 to write the data to Index, then PAC2510 send a response message to confirm.



6. Protocol

6.1. Request:

- Frame ID: 00000623
  - → CAN ID: 600 + 23 (Node ID)
- Data Frame: 2B 17 10 00 A5 5A 00 00
  - → CMD: 2B (2byte data length)
    Index: 1017
    Sub-index: 0
    2-byte data: A5 5A

➔ Data 0xA5 5A is written to Index: 0x1017

6.2. Confirm:

- Frame ID: 000005A3
  - → CAN ID: 580 + 23 (Node ID)
- Data Frame: 2B 17 10 00 A5 5A 00 00
  - → CMD: 60 (OK)
    Index: 1017
    Sub-index: 0
    Byte4-7: 0

4.2.2. Initiate SDO Upload

Client Server

Init SDO Upload request

CAN ID	Data						
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
600+NodeID	0x40	<IDX>		<SUBIDX>	0x00	0x00	0x00	0x00

Client Server

Init SDO Upload confirm (OK)

CAN ID	Data						
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
580+NodeID	<CMD>	<IDX>		<SUBIDX>	<Data>			

Init SDO Upload confirm (error)

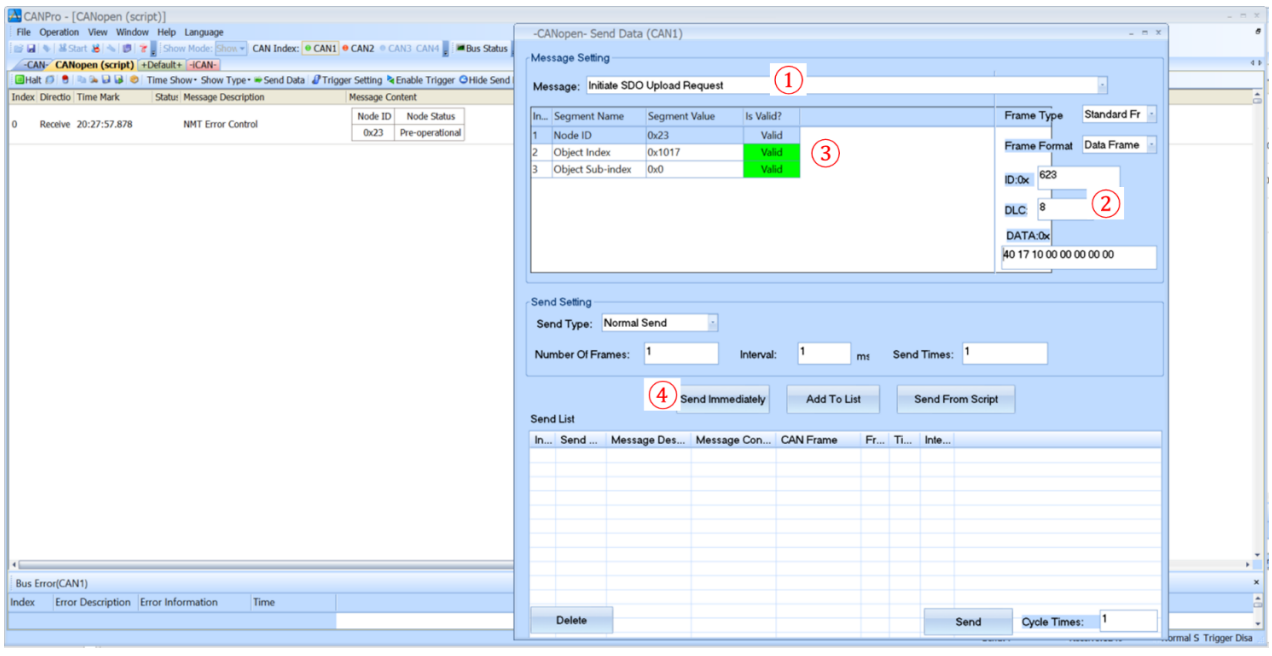
CAN ID	Data						
	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
580+NodeID	0x80	<IDX>		<SUBIDX>	<ERROR CODE>			

The length of the data is encrypted in the <CMD> of the answer:

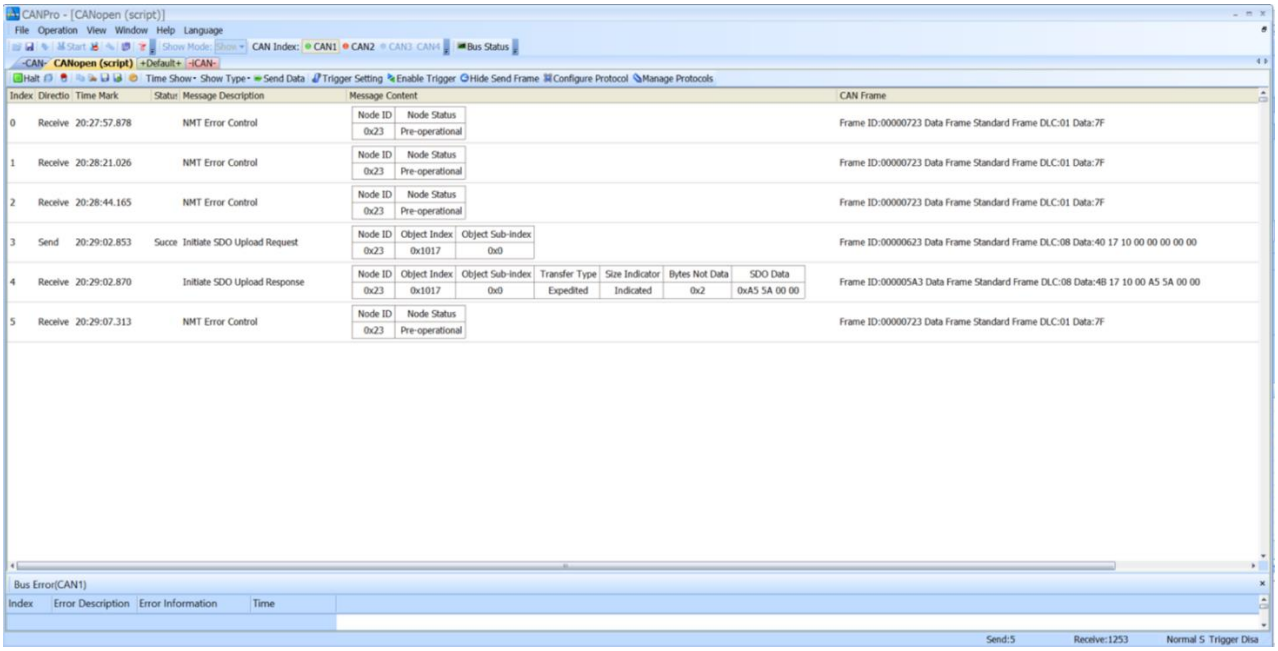
1 byte data length:	4F h
2 byte data length:	4B h
3 byte data length:	47 h
4 byte data length:	43 h

Similar with *Start Remote Control* for step 1-3.

4. Select [Send Data] → 4.1. In the send window Select Message [Initiate SDO Upload Request] → 4.2. Set DLC: 8 → 4.3. Set Note ID: 0x23 - Index: 0x1017 (2bytes) - Sub-index: 0 → 4.4. Send Immediately.



5. CANalyst send a request upload to PAC25140 to read data on the Index, then PAC2510 send a response data to Confirm.



6. Protocol

6.1. Request:

- Frame ID: 00000623
  → CAN ID: 600 + 23 (Node ID)
- Data Frame: 40 17 10 00 00 00 00 00
  → CMD: 40 (fixed)
  Index: 1017
  Sub-index: 0
  Byte4-7: 0

6.2. Confirm:

- Frame ID: 000005A3
  → CAN ID: 580 + 23 (Node ID)
- Data Frame: 4B 17 10 00 A5 5A 00 00
  → CMD: 4B (2byte data length)
  Index: 1017
  Sub-index: 0
  2-byte data: A5 5A

➔ Data 0xA5 5A is written to Index: 0x1017, and can read by Upload Request.

Contact Information

For the latest specifications, additional product information, worldwide sales and distribution locations:
Web: www.qorvo.com
Tel: 1-844-890-8163
Email: customer.support@qorvo.com

Important Notice

The information contained herein is believed to be reliable; however, Qorvo makes no warranties regarding the information contained herein and assumes no responsibility or liability whatsoever for the use of the information contained herein. All information contained herein is subject to change without notice. Customers should obtain and verify the latest relevant information before placing orders for Qorvo products. The information contained herein or any use of such information does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other intellectual property rights, whether with regard to such information itself or anything described by such information. **THIS INFORMATION DOES NOT CONSTITUTE A WARRANTY WITH RESPECT TO THE PRODUCTS DESCRIBED HEREIN, AND QORVO HEREBY DISCLAIMS ANY AND ALL WARRANTIES WITH RESPECT TO SUCH PRODUCTS WHETHER EXPRESS OR IMPLIED BY LAW, COURSE OF DEALING, COURSE OF PERFORMANCE, USAGE OF TRADE OR OTHERWISE, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.**
Without limiting the generality of the foregoing, Qorvo products are not warranted or authorized for use as critical components in medical, life-saving, or life-sustaining applications, or other applications where a failure would reasonably be expected to cause severe personal injury or death.