CANopenNode Manual

Janez Paternoster
Cesta v Zajčjo Dobravo 14a
Ljubljana
Slovenia, EU
janez.paternoster@siol.net

1. December 2005

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.

Project can be found on Internet page: http://sourceforge.net/projects/canopennode/

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 loosing 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 transfered 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 8 byte wide data object, transmitted from one node to that nodes, 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):** The Network Management objects 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 send in case of error or warning in node.
- **EDS (Electronic Data Sheets):** are text files used in CANopen to describe Object Dictionary implemented in a specific node. They can be used with configuration tools for configure CANopen network.

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

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, 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 (11 bit 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, subindex 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, subindex). 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 1 ms 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 send. 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.
- **Example for Command Panel:** with PIC, numeric keypad and alphanumeric 2*16 LCD display. SDO client. Easy configurable parameters for accessing Object Dictionary variables from any node on network. Password protection. Custom screens. Ready to use.
- **Example for Ethernet to CANopen interface:** with Beck IPC@chip SC1X web controller and SJA1000 CAN controller. HTML interface 1: Status, Heartbeat consumer, NMT master, Receive PDOs, Transmit message, SDO client. HTML interface 2: Easy configurable parameters for accessing Object Dictionary variables from any node on network. HTML interface 3: CAN trace with 1ms time stamp and 64 kb of memory dump. Ready to use.
- **EDS:** Electronic data sheet for each example is included.

1.3 What's new in 1.10

Version 1.10 is final and widely tested version of CANopenNode. It has included all documentation: this Manual, Tutorial and Electronic data sheets. Many working examples are available.

This version has different structure than previous versions. But basically the code in files remains the same. Same is also interface with user application – User.c file. So if you want to port your old CANopenNode project to this new version, best solution is to use structure and files from this version, and copy code from old project to new. This is especially true for CO_OD.c and CO_OD.h files.

If it is not necessary, user should not change files from CANopen folder. For later versions of CANopenNode only those files will need to be upgraded.

2. Design

CANopenNode is curently designed for:

- 8 bit Microchip PIC18FXXX and MPLAB C18 compiler (>=V3.00).
- 16 bit BECK SC1X with Philips SJA1000 and Borland C++ 5.02 compiler.
- It can be translated for use with other compilers and microcontrollers.

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

- **_doc (folder) –** documentation:
  - **Manual.pdf –** This manual in pdf format;
  - **Manual.odt –** This manual in OpenDocument Text format, editable with OpenOffice.org;
  - **Tutorial.pdf –** Tutorial in pdf format;
  - **Tutorial.odt –** Tutorial OpenDocument Text format, editable with OpenOffice.org;
  - **fdl.txt –** GNU Free Documentation License;

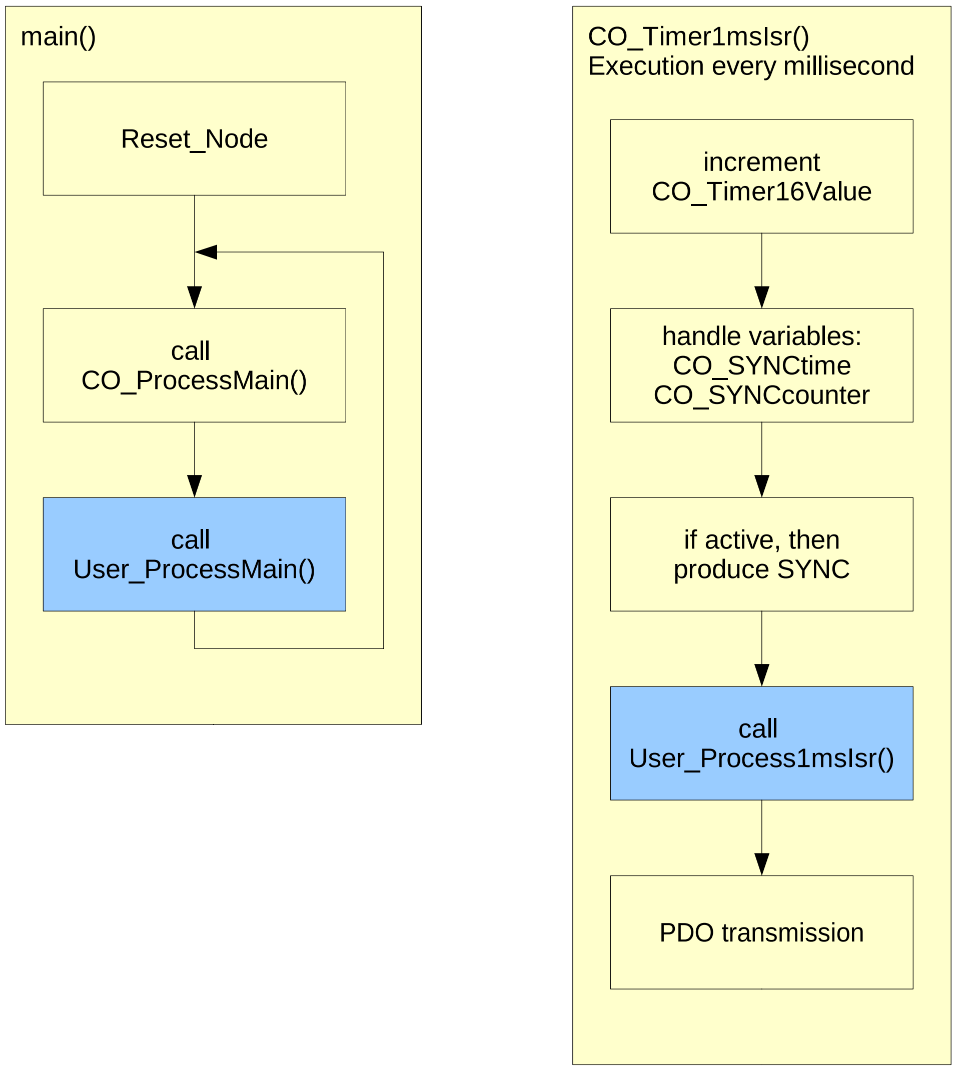
- **_src (folder) –** CANopenNode source code. It is divided into folder CANopen and multiple folders with examples:
  - **lesser.txt –** GNU Lesser General Public License;
  - **readme.txt –** instructions, how to open example project;
  - ***.mcp, *.ide files –** example project files;
  - **CANopen (folder) –** All CANopenNode specific files, except object dictionary. In one project with multiple CANopen devices, this folder can be common for all devices (like library):
    - **CANopen.h –** main header file for CANopenNode. Included are general definitions and other headers;
    - **CO_stack.c –** majority of CANopenNode code;
    - **CO_errors.h –** definitions for all errors, which can occur in program. Also values for emergency error codes and definitions for calculation of the Error register. For user program definitions for user errors can be added;
    - **CO_OD.txt –** short description of object dictionary entries and SDO abort codes;
    - **Subfolders** with microcontroller specific code:
      - **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;
      - **Other** special files;
  - **_blank_project** (folder) – example project, which have no user specific code. works with all microcontrollers:
    - **CO_OD.h –** header for Object Dictionary and main setup for CANopenNode;
    - **CO_OD.c –** variables, verify function and Object Dictionary for CANopenNode;
    - **user.c –** Example functions, that can be used in user program and must be defined.
    - **_blank_project.eds –** Electronic data sheet valid with default configuration;
  - **Other** example project **folders** – see manual and tutorial.

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_ProcessMain` 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 1 ms, overflow occurs, error bit is set and Emergency message is send.
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).

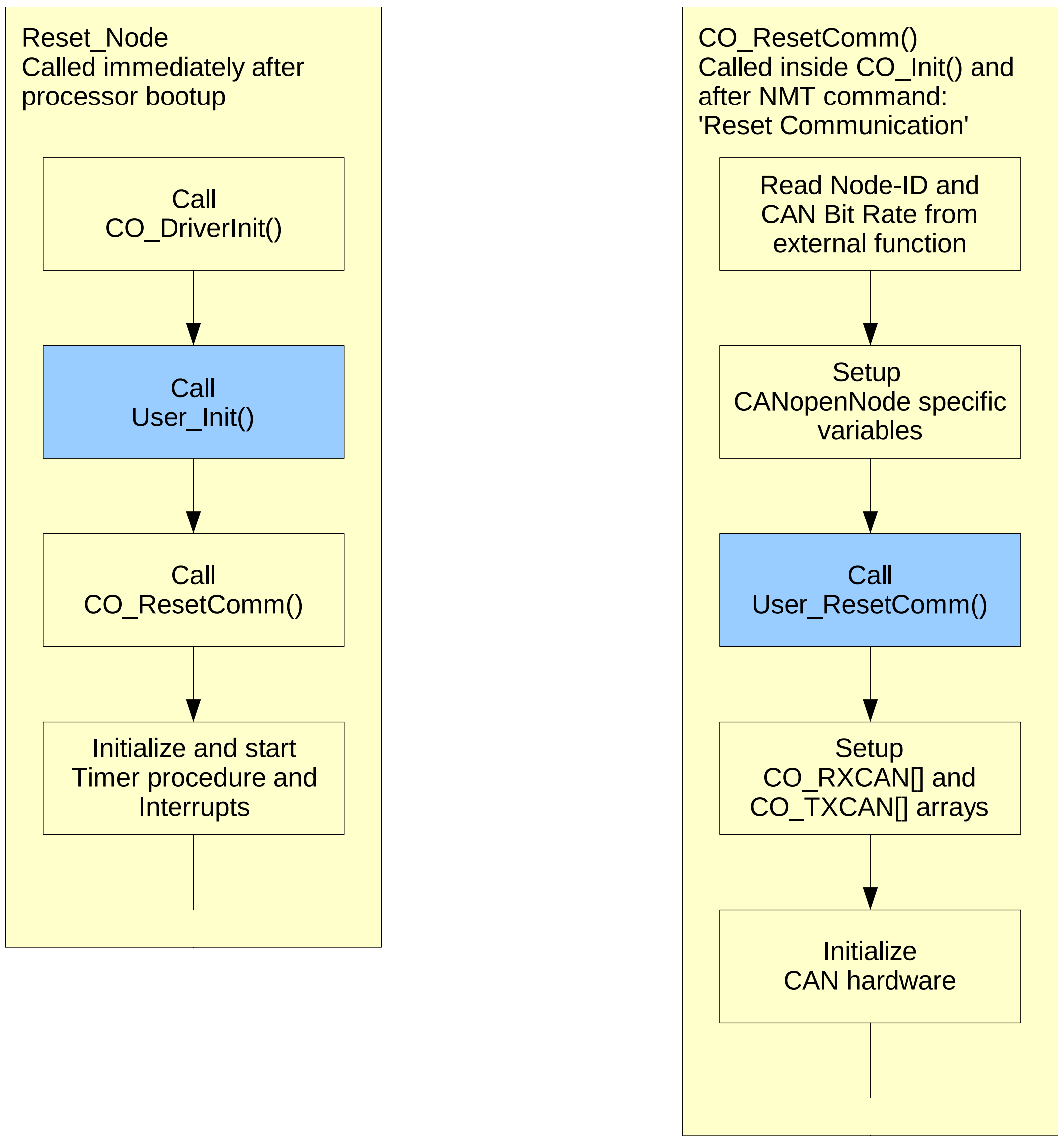

main()
Reset_Node
call CO_ProcessMain()
call User_ProcessMain()
CO_Timer1msIsr()
Execution every millisecond
increment CO_Timer16Value
handle variables: CO_SYNCtime CO_SYNCcounter
if active, then produce SYNC
call User_Process1msIsr()
PDO transmission

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.


Reset_Node
Called immediately after processor bootup
Call CO_DriverInit()
Call User_Init()
Call CO_ResetComm()
Initialize and start Timer procedure and Interrupts
CO_ResetComm()
Called inside CO_Init() and after NMT command: 'Reset Communication'
Read Node-ID and CAN Bit Rate from external function
Setup CANopenNode specific variables
Call User_ResetComm()
Setup CO_RXCAN[] and CO_TXCAN[] arrays
Initialize CAN hardware

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_RXCAN[]` and `CO_TXCAN[]` 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 CANopen.h file.

Type of one element in each array is described below:

```
typedef struct{
   tData2bytes  Ident;
   unsigned int NoOfBytes    :4;
   unsigned int NewMsg       :1;
   unsigned int Inhibit      :1;
   tData8bytes  Data;
}CO_CanMessage;
```

`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_IDENT_WRITE(CobId, RTR)` from CO_driver.h. Message is received if CAN-ID and RTR are matched.

`NoOfBytes` 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.

`NewMsg` 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.

`Inhibit` flag has different meaning. For reception, data from new message received from bus are not copied if (`Inhibit==1` AND `NewMsg==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 Inhibit==1 AND time is outside SYNC window (OD, index 1007h).

`Data` is 8 bytes of CAN data.

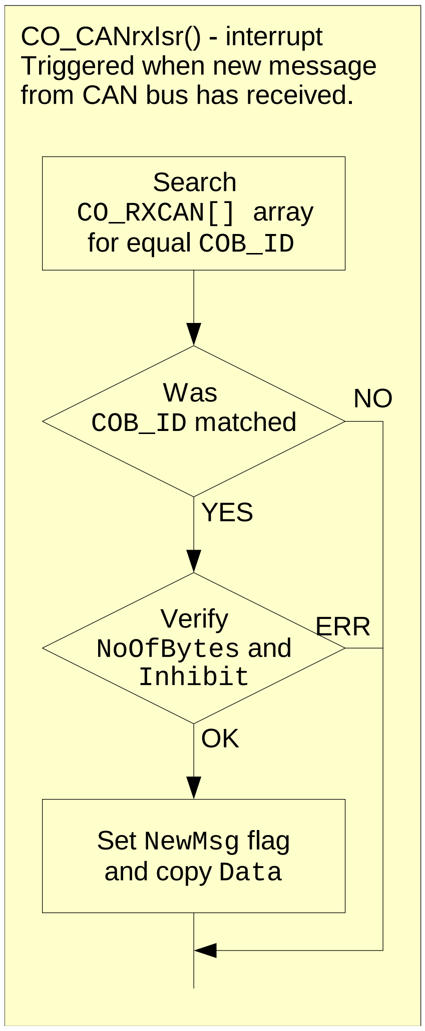
At communication reset first whole array is clearerd and then all values except Data are initialized (inside `CO_ResetComm()` function). This way minimal data copying is achieved, and no other buffers are needed.

2.4.2 Reception of CAN messages

CANopen uses many different CAN identifiers, so hardware filtering of messages can not 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_RXCAN[]` array at startup. When new message arrives from CAN bus, `CO_CANrxIsr()` 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.

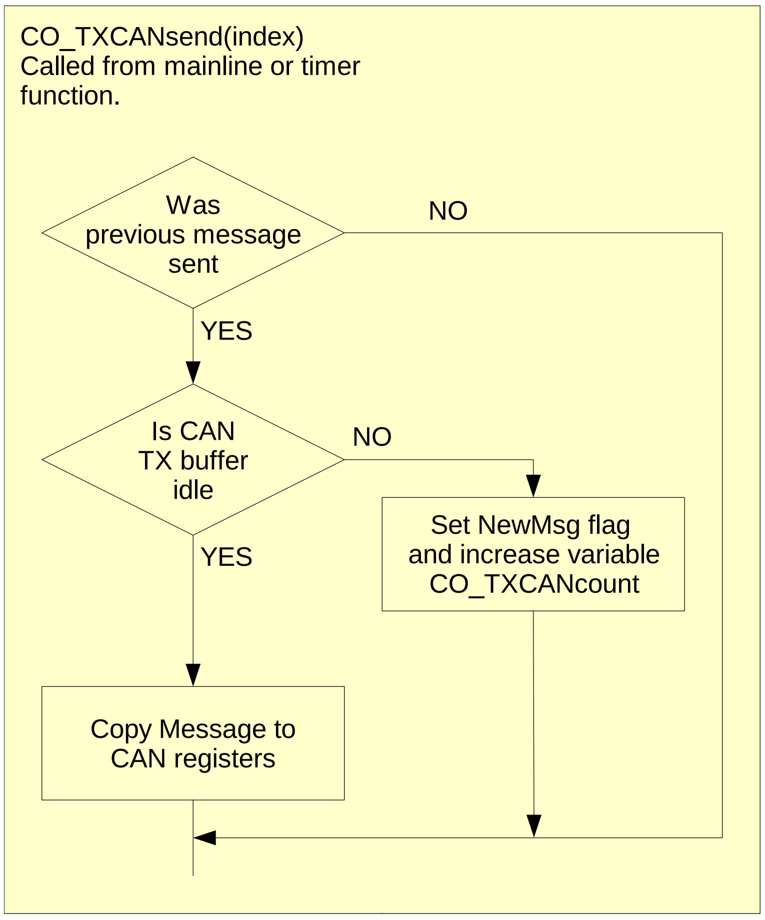
Received messages are then handled inside different functions. Details will be described in next chapters.

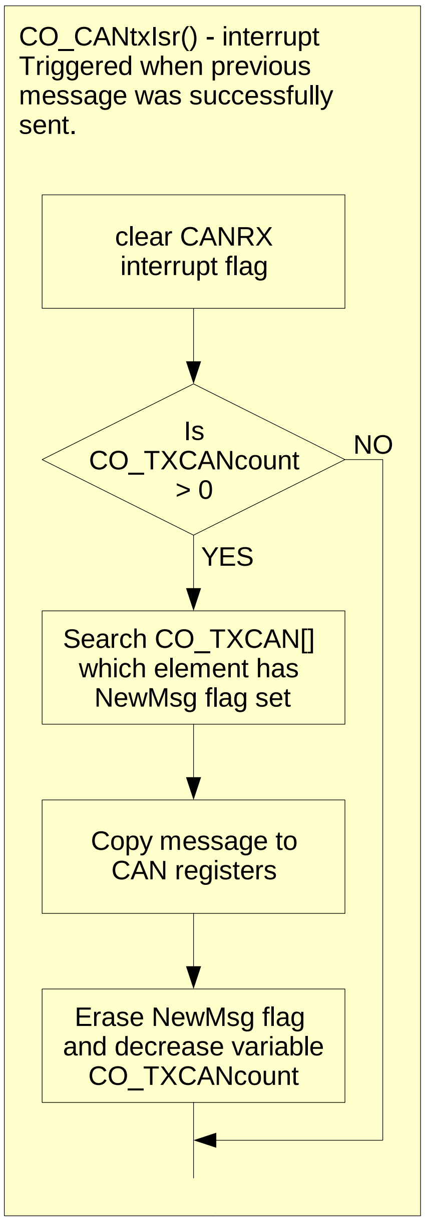

CO_CANrxIsr() - interrupt
Triggered when new message from CAN bus has received.
Search CO_RXCAN[] array for equal COB_ID
Was COB_ID matched
NO
YES
Verify NoOfBytes and Inhibit
ERR
OK
Set NewMsg flag and copy Data

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_TXCAN[]` array element: writes Data and, if necessary, it can also modify other parameters. Then it calls `CO_TXCANsend()` function. If CAN TX buffer is free, message will be send immediately, otherwise it will be marked, and interrupt will send it. Messages with lower `index` (`CO_TXCAN[index]`) will be send first.


CO_TXCANsend(index)
Called from mainline or timer function.
Was previous message sent
NO
YES
Is CAN TX buffer idle
NO
YES
Set NewMsg flag and increase variable CO_TXCANcount
Copy Message to CAN registers


CO_CANtxIsr() - interrupt
Triggered when previous message was successfully sent.
clear CANRX interrupt flag
Is CO_TXCANcount > 0
NO
YES
Search CO_TXCAN[] which element has NewMsg flag set
Copy message to CAN registers
Erase NewMsg flag and decrease variable CO_TXCANcount

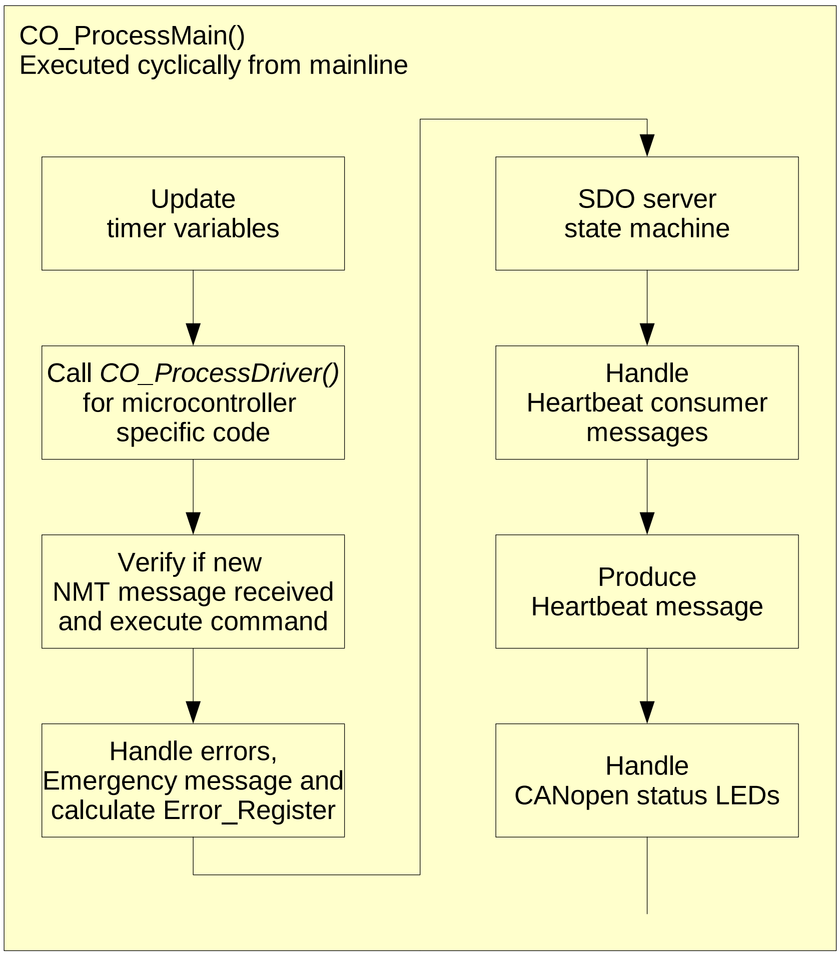
Picture 2.4 – Transmission of CAN messages

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_ProcessMain()` 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 `User_ProcessMain()` is processing. There can also be time consuming and not-time-critical code. Anyway, blocking functions must **not** be used.


CO_ProcessMain()
Executed cyclically from mainline
Update timer variables
Call CO_ProcessDriver() for microcontroller specific code
Verify if new NMT message received and execute command
Handle errors, Emergency message and calculate Error_Register
SDO server state machine
Handle Heartbeat consumer messages
Produce Heartbeat message
Handle CANopen status LEDs

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_ProcessMain()` function. According to command, actions are triggered. NMT master can be easy 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_SYNCtime` and `CO_SYNCcounter` (type unsigned int). CO_SYNCtime is incremented each millisecond and reset to zero, each time SYNC is received/transmitted. 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), `ErrorReport()` function is called, which just sets some variables. When in turn, error handling is processed inside `CO_ProcessMain()` function. All Error handling specific definitions are collected in file `CO_errors.h`.

Following function accepts two parameters:

```
void ErrorReport(unsigned char ErrorBit, unsigned int Code);
```

`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 `CO_ErrorStatusBits` array. If that bit was already set before, nothing happens. This way specific error is reported only the first time, later repeatings are ignored. Opposite function to `ErrorReport()` is `ErrorReset()`, where specific bit is cleared if error is solved somehow.

If new error occurred, procedure inside `CO_ProcessMain()` function verifies new situation. First, Error Register (OD, index 0x1001) is calculated from `CO_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 `ErrorReport()` function. 7th and 8th byte can be user specific.

Each bit in 1 byte long Error Register is calculated from state of `CO_ErrorStatusBits` array. See `ERROR_REGISTER_BITn_CONDITION` macros in `CO_errors.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.

<u>If Error Register is not equal to zero, node is prevented to be in Operational state, and PDOs can not be transmitted or received.</u>

If macro `ODD_ERROR_BEH_COMM` in `CO_OD.h` file is set to 1, Communication error bit in Error register wont 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 can not 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 won't be processed and emergency will be send first time. If mapping parameters are disabled by disabling macro CO_PDO_MAPPING_IN_OD, 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. <u>If bit 31==1, PDO is not used.</u>

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.

<u>PDO transmission in CANopenNode</u>

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 can not 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). Type of one array element is `tData8bytes`. Parenthesis () instead [] are used for array, because of macro syntax.

To send PDO manually, just call the following function:

```
	char CO_TPDOsend(unsigned int index);
```
where `index` 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). Type of one array element is `tData8bytes`. Parenthesis () instead [] are used for array, because macro syntax.

`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_SYNCtime == 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_wait()` and `CO_SDOclientUpload_wait()`.

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. Type of one array element is `unsigned char`. Values are defined in `CANopen.h`. Parenthesis () instead [] are used for array, because macro syntax.

2.6.8 User defined Communication objects

Besides standard COBs, user can define his own, for example NMT master message. Usage for rx (tx) messages:

1. set `CO_NO_USR_CAN_RX` (`CO_NO_USR_CAN_TX`) definition to more than zero.
2. Initialize `CO_RXCAN[CO_RXCAN_USER + i]` (`CO_TXCAN[CO_TXCAN_USER + i]`) array (). 0<`i`< `CO_NO_USR_CAN_RX` (0<`i`< `CO_NO_USR_CAN_TX`). Don't set `NewMsg` flag!
3. Read received data from `CO_RXCAN[CO_RXCAN_USER + i]`, when NewMsg bit from that array is 1. (Write data to `CO_TXCAN[CO_TXCAN_USER + i]` and call `CO_TXCANsend(CO_TXCAN_USER + i)` to send message).

Example for NMT master message (don't forget point 1.):

```
CO_TXCAN[CO_TXCAN_USER].Ident.WORD[0] = CO_IDENT_WRITE(0/*COB-ID*/, 0/*RTR*/);
CO_TXCAN[CO_TXCAN_USER].NoOfBytes = 2;
CO_TXCAN[CO_TXCAN_USER].Inhibit = 0;
CO_TXCAN[CO_TXCAN_USER].Data.BYTE[0] = NMT_RESET_COMMUNICATION;
CO_TXCAN[CO_TXCAN_USER].Data.BYTE[1] = 0;
CO_TXCANsend(CO_TXCAN_USER);
```

2.7 Object Dictionary

One of the most powerful feature 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, subindex, 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, subindex 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 PIC18Fxxx resets during write data can be corrupted;
   - In PIC18Fxxx no RAM is used;
   - In object dictionary it is marked with attribute ATTR_ROM.
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 ODE_EEPROM structure);
   - At chip initialization values are read from EEPROM;
   - At run time background routine compares RAM and EEPROM and saves byte by byte if different (PIC18Fxxx), or data are saved after power fail signal (SC 12);
   - Classic writing to variable by user program;
   - Possible problem: if microcontroller resets during write: multi byte variable can be corrupted (PIC18Fxxx).

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, attribute, length and pointer to variable;
- Attributes:
  - `ATTR_RW` - read-write;
  - `ATTR_WO` - write-only;
  - `ATTR_RO` - read-only (TPDO may read from that entry);
  - `ATTR_CO` - read-only, constant;
  - `ATTR_RWR` - read/write on process input (TPDO may read from that entry);
  - `ATTR_RWW` - read/write on process output (RPDO may write to that entry);
  - `ATTR_ROM` - variable is saved in retentive memory (ROM variable);
  - `ATTR_ADD_ID` - add NODE-ID to variable value (sizeof(variable)<=4);
- When SDO server wants to find entry (with specific index and subindex) inside object dictionary, it calls `CO_FindEntryInOD()` function. 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 verify 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. Hardware and compiler specific files

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;
- Hardware macros: pins for green and red LED, etc.
- Different software and interrupt enable/disable macros;
- Function for reading Node-ID and Bit-rate 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 PIC18 with Microchip C18

PIC18Fxxx is family of 8-bit microcontrollers from [Microchip]. CANopenNode is designed for those with integrated CAN module. It is tested with PIC18F458.

Compiler used is Microchip MPLAB C18 >= V3.00.

Additional files included:

- **memcpyram2flash.h** – header for `memcpyram2flash.c`;
- **memcpyram2flash.c** – PIC18Fxxx specific function, that copies contents of RAM variable into Flash program memory space. It takes 20-30 ms for complete erase and write cycles. Function also uses 64 bytes of stack for data backup. During this time microcontroller is frozen and must not lose power supply. Function is used, when Object Dictionary ROM variables are written;
- **18f458i.lkr** – default linker script from compiler.
- **CONFIG18f458.c** – configuration bits for microcontroller.

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.

3.1.1 Hardware design

PIC18Fxxx device with integrated CAN module and CAN transciever. Green LED is connected to RB4 and Red LED is connected to RB5 (can be changed).

3.1.2 Resources in PIC18F458

Resource	Description
CAN Module	
Port B: RB2, RB3	For CANTX and CANRX lines.
INT2	External interrupt2 – it is multiplexed with RB2
Port B: RB4, RB5 (default)	Pins used for indicator LED diodes. Pins can be changed or not used.
TIMER2 module	Used for 1ms timer interrupt.
PWM module	It is free for use, except PWM frequency is set with initialization of TIMER2.
Interrupts	Both, Low and High priority Interrupts are used. User may add his own interrupt sources.
Program memory	17 kB (52%) by default*
RAM memory	662 bytes (43%) by default*

* default settings are based on _blank_project: 4 RPDO, 4 TPDO, 4 Heartbeat consumer, 8 Error fields, Max OD entry size = 20 bytes, flash and EEPROM write enabled, PDO parameters and mapping in object dictionary, map size = 8. No optimization. Stack and SFR registers included in calculation. MPLAB C18 compiler.

3.1.3 Characteristics in PIC18F458

Some measurements were made to check performance of the driver.

Settings:

- quartz for PIC: 10MHz, PLL enabled, that gives 40 MHz clock frequency, and 10 MIPS;
- CAN baud rate: 1Mbps;
- 4 transmit PDO, 4 receive PDO, 4 heartbeat consumers;
- reception of 11 different CAN messages with different identifiers, no hardware filtering used.

	Value	Description
Low Interrupt Latency	19 µs	Interrupt context saving. At interrupt end context are restored and additional latency must be considered.
Low Interrupt – transmit message	Max 30 µs	
Low Interrupt – timer	6 µs	
Low Interrupt – prepare synchronous transmit PDOs 4x.	55 µs	
High Interrupt Latency	1,5 µs	Interrupt context saving. At interrupt end context are restored and additional latency must be considered. High interrupt is used to receive CAN messages. If number of different CAN IDs is less or equal to 6, hardware filtering is used.
High Interrupt – process message	Max 14 µs	

	Value	**Description**
main() - average time	40 µs	
Process SDO – write to RAM or read.	160 µs	CO_OD array has ordered indexes and subindexes, so search is fast even for large object dictionary.
Process SDO – write to ROM.	20 ms	During this time microcontroller is frozen. If this is the problem, use RAM variables inside ODE_EEPROM structure.

3.2 BECK SC1x + SJA1000

Beck IPC@CHIP SC1x is a single chip embedded controller and includes a 16bit 186-Processor, RAM, Flash-Disk, an Ethernet-Controller, RT operating system, web server, etc. Together with Philips SJA1000 CAN controller is capable to be also a CANopenNode device.

Home page for IPC@CHIP is: http://www.beck-ipc.com/ipc/index.asp?sp=en

Compiler used is Borland C++ 5.02. Required library C-Lib is available from link above.

3.2.1 Memory strategy for retentive variables

User does not need to understand details of this chapter, because retentive variables works automatic.

ROM variables are implemented with file system. Variables are read in the beginning from file (for name of the file see table below). On change of ROM variable, file is rewritten by default. For different behavior (recommended) see table below .

File format: it is text file with one Object Dictionary entry in each line. Order is not important.

```
IIIISSFF:DDDD...
```

or

```
IIII SS F : DD DD ...
```

`IIII` is index in hex format (must be 2 bytes), `SS` is subindex in hex (must be 1 byte), `FF` is format of data (must be 1 byte, 0=hex, 1=ASCII, 2=signed, 3=unsigned), colon is obligatory, `DD` are data in specified format. **Data length must match to length of corresponding entry in OD**.

Format of data read from file *(CO_sscanfTextToData() function)*

0. `CO_IO_DataFormatHEX` – One or two Hex digits in input string represents byte, most significant bytes comes first. (Data after parsing are in little endian format, as is defined in CANopen.) One letter between spaces (including tabs) represent 1 byte, two letters also (for example '5 F' are two bytes – 5 and 15 in decimal; '5F' is one byte – 95 in decimal; '340356FEADC123' is same as '34 03 56 FE AD C1 23' is same as '34 3 56 FE ADC123'). Otherwise spaces are ignored. All characters after nonXdigit or nonspace character are ignored.
1. `CO_IO_DataFormatASCII` – Data as string between colon and newline character. Length must be the same as in OD. (Note: If OD variable is declared like `char ODE_var1[] = “some_string”;` and then put into CO_OD like `OD_ENTRY(... , ODE_var1)`, then length of variable on CO_OD includes NULL character in the end of string. In this case length is 12. *CO_sscanfTextToData()* does not include NULL character in its length information.)

2. `CO_IO_DataFormatSIGNED` – Data are converted to integer or long with following rules: an optional string of tabs and spaces, an optional sign, a string of digits. The first unrecognized character ends the conversion. There are no provisions for overflow (results are undefined).
3. `CO_IO_DataFormatUNSIGNED` – Data are converted to unsigned integer or unsigned long with following rules: an optional string of tabs and spaces, a string of digits. The first unrecognized character ends the conversion. There are no provisions for overflow (results are undefined).

Eeprom variables are made with Non-Volatile Data, feature of the IPC@CHIP.

After building the project, binary file is created, for example CANopen.exe. It is then downloaded into IPC@CHIP and can be executed. Optional command parameters are:

Command	Result
-OD_ROM_FileName *filename*	Name of file, where values for ROM variables in Object Dictionary are stored. Default filename is "CANopen.ini". File path may also be included. If file doesn't exist on startup or on writing to ROM variable, new file is created from all ROM variables in Object dictionary.
-OD_ROM_MaxFileSize *size*	By default this value is set to 0. This means, that file with all ROM variables from object dictionary will be rewritten each time, SDO communication changes any ROM variable. If size is greater than 0, new entry is just added to the end of the file. This way also history of changes is available. If file exceeds length defined by *size* (in bytes), old file is renamed to *.bak and new file is created from all ROM variables in Object dictionary. It is recommended to set *size* to value, greater than size of newly created file. Size depends on: memory available, how often variables are changed, history we want. Too small or too large file will slow down either write to object dictionary or file read at startup.
-OD_ROM_TimeStamp *option*	If option is different than zero, time stamp will be added into file each time variable will be written.
-? or -help	Print help

This example runs CANopenNode. ROM variables are(will be) stored on external disk, maximum file size will be 10000 bytes, after that file will be renamed to *.bak and new, fresh file will be created:

```
CANopen.exe - OD_ROM_FileName B:\CANopen.ini -OD_ROM_FileSize 10000
```

3.2.2 Hardware design

Hardware is designed according to schematic:

http://www.beck-ipc.com/ipc/download/pdf.asp?id=699&sp=en

Additional Green LED is connected to PIO 0 and Red LED is connected to PIO 1 (can be changed).

For Eeprom variables, power fail signal and large enough capacitor must be used.

Electrical connections between SC12 and SJA1000 are fast, so PCB design must be careful!

For example see next chapter.

4. Examples and Object Dictionary

Main files in Examples are:

- **CO_OD.h** – definitions for CANopenNode, default values for Object Dictionary;
- **CO_OD.c** – variables, verify function and Object Dictionary for CANopenNode;
- **user.c** – Example functions, that can be used in user program and must be defined.
- ***.eds –** Electronic data sheet for CANopen configuration tool.

These files contain:

- Macros for Setup CANopenNode;
- Global variables for Object dictionary;
- Default values for variables;
- Object dictionary, where address, index, subindex and attributes of above variables are stored;
- Verify function `CO_OD_VerifyWrite()`, which verifies value of variable, before it is written to object dictionary;
- Functions, from where user code can be executed: `User_Init()`, `User_ResetComm()`, `User_ProcessMain()` and `User_Process1msIsr()`.

4.1 _blank_project

Compete CANopenNode. User functions are empty. It is microcontroller independent.

4.2 Example_generic_IO

Example for implementing standard CANopen generic Input/Output module (CiADS401, Digital inputs, outputs, Analog inputs, outputs). It is based on _blank_project example. It is microcontroller independent. In User.c file must be added definitions for hardware. See Tutorial.

4.3 Panel with PIC+LCD+keypad

It is an example for user panel – command interface. It is made of PIC18f458, alphanumeric LCD display 2x16, numeric keypad 3x4 and CANopenNode. All functions are nonblocking.

Additional files are:

- **LCD+Keyb.c/h** – basic functions for LCD and keyboard. Debouncing is used for keyboard. It is also scanned, how long specific key is pressed;
- **MenuOnLCD.c/h** – functions for implementing menus;
- **user.c** – code for menu: basic screen, screens for setting parameters (SDO client access to Object dictionaries on all nodes), password screen (screens after that screen are accessed only to administrator), screens for universal SDO client, many other possibilities;
- **CANopenPanel.hex** – compiled code for use with tutorial (default Node-ID=0x15, bitrate = 125 kbps, oscillator = 32 MHz).

Project must be compiled with full optimization. For using see the Tutorial. Schematic is in file `_src\examples\panel_with_PIC+LCD+keypad\PIC+LCD+keypad.sch.pdf`.

`panel_with_PIC+LCD+keypad.eds` file is Electronic Data Sheet, which is valid with default settings in `CO_OD.h` file.

Keyboard used looks like this:

1	2	3
4	5	6
7	8	9
*	0	#

***** is like *escape* : when pressed, it escapes from input field, goes to beginning of menu entry or goes to basic screen;

is like *enter* : it reads the value under one menu, enters into input field or confirms input;

7 and **9** are like ← and → : when not inside input field, keys are used as previous and next menu entry;

0 ... **9** are used in input field for typing numbers. If input field is hexadecimal, long press on 0 ... 5 gives numbers A ... F.

Menus can be very flexible made. Some elements are:

- **Input field** (hex or dec): during user input, cursor is blinking, otherwise not;
- **Screen MenuParameter**: complete screen with background text, NodeID-Index-Subindex location of parameter, min(max) value of parameter, etc. It uses decimal input field. Here is an example screen for `Producer_Heartbeat_Time` variable on node 5:

```
Prod Heartbeat 5
10▓  [ms]
```

- **Univesal SDO client** uses three screens. All input fields are hexadecimal and are marked as `~~strikethrough~~`:
  1. Setup screen has input fields for NodeID, Index, Subindex and length of variable:
     ```
     NodeID:~~NN~~ Len:~~LL~~
      Idx:~~IIII~~ Sub:~~SS~~
     ```
  2. Read screen has information about above four variables and after pressing # reads the value and prints it in hex format. In upper right corner has 'RD'.
     ```
     NN LL IIII SS RD
     ```
  3. Write screen is similar, except in upper right corner is 'WR'. After pressing # hex input field is activated.
     ```
     NN LL IIII SS WR
     ```

4.4 Web_interface_with_SC1x

It is an example for advanced user panel – command interface. Hardware is made of BECK IPC@Chip SC13 and SJA1000 CAN controller. Software used is multitasking RTOS delivered with IPC@Chip and CANopenNode.

IPC@Chip includes also ethernet controller and http server, so it can be connected to ethernet and accessed with web browser (like Mozilla Firefox or Internet Explorer on PC). CANopenNode generates dynamic html pages – contents depends on different variables and user input.

Additional files are:

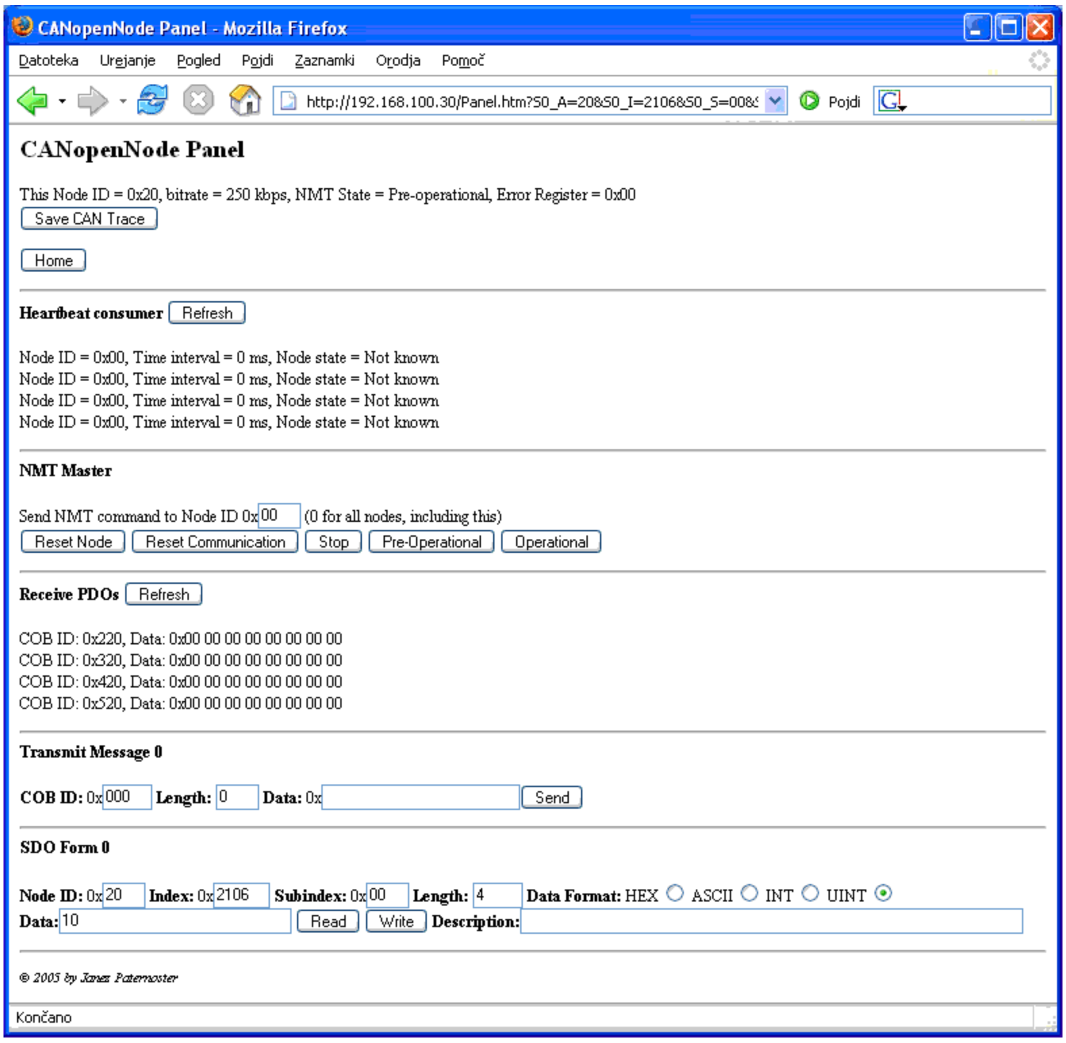
- **CO_CGI.c –** function that writes dynamic html pages;
- **CANopenl.exe –** binary for use with tutorial (default Node-ID=0x20, bitrate = 125 kbps).

For schematic see chapter 3.2.

`web_interface_with_SC1x.eds` file is Electronic Data Sheet, which is valid with default settings in `CO_OD.h` file.

Three CGI pages are avilable:

- **Panel.htm** – Main command panel: Node state, Heartbeat consumer, RPDO, Transmit message, SDO client;
- **Param.htm** – Configurable parameters: SDO client with access to Object dictionaries on all nodes. Parameters are defined in Object dictionary, index 0x2200+, sub-indexes 0..7:
- **cantr.dmp** – CAN trace – binary file, which contains up to 64 kb of latest CAN messages with time stamp (with 1ms accuracy). Cantr.dmp file can be decoded on PC with cantrace.exe.


CANopenNode Panel - Mozilla Firefox
Datoteka Urejanje Pogled Pojdi Zaznamki Orodja Pomoč
http://192.168.100.30/Panel.htm?S0_A=20&S0_I=2106&S0_S=00&
Pojdi

CANopenNode Panel

This Node ID = 0x20, bitrate = 250 kbps, NMT State = Pre-operational, Error Register = 0x00

Save CAN Trace

Home

Heartbeat consumer Refresh

Node ID = 0x00, Time interval = 0 ms, Node state = Not known
Node ID = 0x00, Time interval = 0 ms, Node state = Not known
Node ID = 0x00, Time interval = 0 ms, Node state = Not known
Node ID = 0x00, Time interval = 0 ms, Node state = Not known

NMT Master

Send NMT command to Node ID 0x 00 (0 for all nodes, including this)

Reset Node | Reset Communication | Stop | Pre-Operational | Operational

Receive PDOs Refresh

COB ID: 0x220, Data: 0x00 00 00 00 00 00 00 00
COB ID: 0x320, Data: 0x00 00 00 00 00 00 00 00
COB ID: 0x420, Data: 0x00 00 00 00 00 00 00 00
COB ID: 0x520, Data: 0x00 00 00 00 00 00 00 00

Transmit Message 0

COB ID: 0x 000 **Length:** 0 **Data:** 0x Send

SDO Form 0

Node ID: 0x 20 **Index:** 0x 2106 **Subindex:** 0x 00 **Length:** 4 **Data Format:** HEX ○ ASCII ○ INT ○ [✓ UINT]
Data: 10 Read Write **Description:**

© 2005 by Janez Paternoster

Končano

Picture 4.1 – Browser access to CANopenNode on SC12

4.5 Tutorial

With tutorial examples, all features of CANopenNode are shown. See Tutorial.pdf.

5. Conclusion

Standardized CANopen protocol is a step towards more quality, reliable and standardized devices. Automation tasks are now easier to implement. Reliable and simple network with 8-bit microcontrollers gives the possibility to make advanced decentralized control system with smart, simple and thus more reliable devices.

Besides networking, CANopenNode gives the devices the necessary features, that makes CANopen just more simple and useful. Everything necessary to make a device is: setup CANopen communication; use custom variables (RAM or Flash or Eeprom) and give them into object dictionary; write code, which does something with those variables, microcontroller resources and CANopen communication objects. It's also flexible, so custom code and also custom communication objects is not a problem.

CANopenNode is light, so enough memory and time in microcontroller remains free for custom tasks.

CANopenNode can also be used in master devices, what is shown in two examples, which shows advanced control over the network.

6. Literature

[CANopenBook]

Olaf Pfeiffer, Andrew Ayre, Christian Keydel “Embendded Networking with CAN and CANopen”, RTC Books, 2003, “http://www.canopenbook.com/”

[CiA]

“http://www.can-cia.org/canopen/”

[CiADS301]

“CANopen Application Layer and Communication Profile”, CiA Draft Standard 301, Version 4.02, free access

[CiADR303-3]

“CANopen Indicator Specification”, Draft Recommendation 303-3, Version 1.0

[CiADS401]

“CANopen Device Profile for Generic I/O Modules”, Draft Standard 401, Version Version 2.1”

[esacademy]

“http://www.esacademy.com/myacademy/” - Online training.

[Microchip]

http://www.microchip.com

[Beck-IPC]

http://www.beck-ipc.com/ipc/index.asp?sp=en

[CANchkEDS]

“http://www.vector-informatik.com/english/products/canchkeds.php” - EDS checker.

[sourceforge]

“http://sourceforge.net/” - Many useful open source projects