Application Note – Interfacing with CAN bus

Table of Contents

contact@wattandwell.com
+33 1 75 95 11 50

usa@wattandwell.com
+1 346-223-0379

wattandwell.com

Revision date: 01/2021

1 Introduction

This document describes CANopen protocol structure and configuration for MPU series chargers. The main purpose is to explain how to build necessary CAN messages to control Modular Power Unit without employing any specific CANopen stack.

MPU series chargers are 25 kW power units where the parallelization of several chargers allows to modulate the delivered power. Hence, CANopen protocol is suitable for fast communication with several power units to be controlled through a CANopen master.

MPU series are dotted with a graphical user interface that can play either the role of a master or a CAN bus sniffer. Both functionalities could be used to debug communication issues if another master than Watt&Well GUI is used. In this document, test examples are given for MPU 25 kW unidirectional charger where the master is run from Python script.

Common terms

MPU	Modular Power Unit
G2V	Grid-to-Vehicle
GUI	Graphical User Interface
CAN	Control Area Network
CANopen	Communication protocol to open and communicate with the Control Area Network
EDS	Electronic Data Sheet
Index	4-digit hexadecimal code used to identify an object: 16-bits
Sub-index	Decimal code to further identify object's parameters: 8-bits
OD	Object Dictionary
Object	Communication message
PDO	Process Data Object
RPDO	Receive PDO
TPDO	Transmit PDO
SDO	Service Data Object
NMT	Network Management

2 References and Required Tools

2.1 Documents

Reference	Document Title
MPU-R3-500-63-FD User Guide	**Modular Power Unit - User guide**
MPU-R3-500-63-FD Datasheet	**MPU-25 Datasheet**

2.2 CAN interface

For PC/CAN interface, it is recommended to use one of the listed below transceivers:

- **National instruments:** NI USB-8473 or NI USB-9861
- **IXXAT:** USB-to-CAN V2 compact
- **Kvaser:** Leaf Light V2

2.3 Communication development environment

In this document Python 2.7 is used to set up communication examples. It can be downloaded from : https://www.python.org/downloads/release/python-2717/.

The library ***canopen 1.0.0*** is required. Installation steps are listed below:

- Install *pip* following instructions from: https://pip.pypa.io/en/stable/installing/
- Install *canopen* library by running the following in command-line tool

```
$ pip install canopen
```

2.4 GUI environment

The GUI is compatible with National instruments interface and Windows 10/7/Vista/XP/200.
NI-CAN drivers must be installed. They can be downloaded from : http://www.ni.com/download/ni-can-18.5/8074/en/

User is referred to GUI user guide documents ***Modular Power Unit - User guide*** for further information.

2.5 Monitoring software

To spy the CAN bus for test or debug purposes, it is recommended to install the following software:

- **National instruments:** NI MAX (Measurement and Automation Explorer) is included in NI-drivers package.
- **IXXAT:** canAnalyser (https://www.ixxat.com/support/file-and-documents-download/demo-software-tools)

2.6 Network configuration and CAN wiring

Physical CAN network must be equipped with 2 termination resistors of 120 Ω each. The smallest CAN network is composed of 2 nodes; the GUI node (master) and the BMPU node:

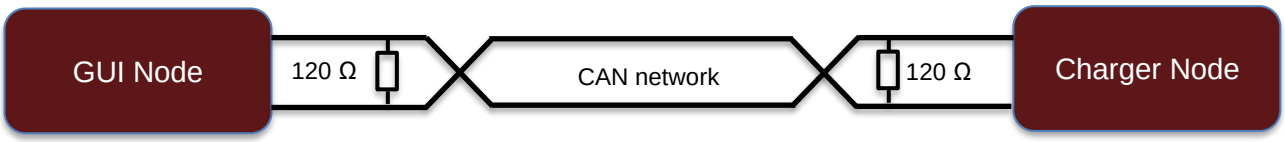


Figure 1 Network nodes

2.7 CAN wiring

For CAN wiring, refer to MPU datasheet ***MPU-25 Specification*** on Interfaces section.

2.8 CAN node ID

For all devices a unique node ID must be selected. MPU-25 takes its CAN **address** at boot based on an addressing connector on the front panel.

Address	CAN ID
1 (001)	80
2 (010)	81
3 (011)	82
4 (100)	83
5 (101)	84
6 (110)	85
Default (000 or 111)	86

Values 0 and 1 on address refer respectively to 0V and 24V on the corresponding connector pin.

If no connector is connected to the charger, the default node ID is 86 (0x56).

Message frame IDs are defined in as: **frame ID = offset ID + node ID**

Example: TPDO0 offset is **0x180** and node ID is **0x56** then TPDO0 ID will be **0x1D6**
=0x180+0x56

2.9 Messages description

For messages description, user is referred to MPU25 specification CAN communication section. In the same section, status word, fault word and data definition are presented. Taking note of CAN message description is required to understand the following section.

The user may also refer to CANopen literature for detailed description of CANopen specific messages as PDO (Processes Data Objects), SDO (Service Data Objects) …etc.

3 Communication tests

The master node ID is 1 and charger node ID is 80 (address set to 001).

3.1 Communication using Python script

Follow the instructions step by step to set up a correct CAN communication using Python script

Step 1 **Test setup**	Set the testbench : • Supply 24V to MPU-25 • Connect Emergency shutdown (24V) • Connect the addressing connector in such way to obtain the address 80 (001) • Connect CAN transceiver to MPU-25 CAN RJ45 connector and to a PC • Connection to AC power is not required and not recommended for communication tests. However, without AC power the charger will go to fault state when Charging mode is requested.
Step 2 **Network configuration**	Start new Python script : • MPU-25 electronic datasheet (pu.eds) must be in the same folder as Python script. Otherwise, file path must be specified in the script when EDS is needed. • Import required libraries

	`import canopen`
	`import sys`
	`import os`
	`import traceback`
	`import time`

	• Configure network and connect

	`try:`
	`  # Start with creating a network representing one CAN bus`
	`  network = canopen.Network()`
	
	`  # Specifiy CAN tranceiver type, CAN channel and baud rate`
	`  network.connect(bustype='ixxat', channel=0, bitrate=500000)`
	`  # network.connect(bustype='nican', channel='CAN0', bitrate=500000)`
	
	`  # Specifiy node address  and the corresponding Electronic Datasheet`
	`  network.add_node(80, './pu.eds',upload_eds=False)`
	`  node = network[80]`
	
	`  # Check network`
	`  network.check()`

	• Send Master Heartbeat message (ID 701) for bootup (MasterStatus=0)
	`  # Master boot up message (MasterStatus =0)` `  network.send_message(0x701, [0x0])`
	• Read PDO configuration from node

	`  # Read PDO configuration from node`
	`  node.tpdo.read()`
	`  node.rpdo.read()`

	• Change master state de operational (MasterStatus=5)
	`  # Change master state to operational (NMT start)`

	network.send_message(0x701, [0x5])
	• Check slave heartbeat

	# Check slave heartbeat
	node.nmt.wait_for_heartbeat()
	assert node.nmt.state == 'OPERATIONAL'

Step 3 Sync message	• Send sync message with a selected period

	# Transmit SYNC every 100 ms
	network.sync.start(0.1)

Step 4 Sending RPDOs	• Configure RPDO0 message data to be sent.

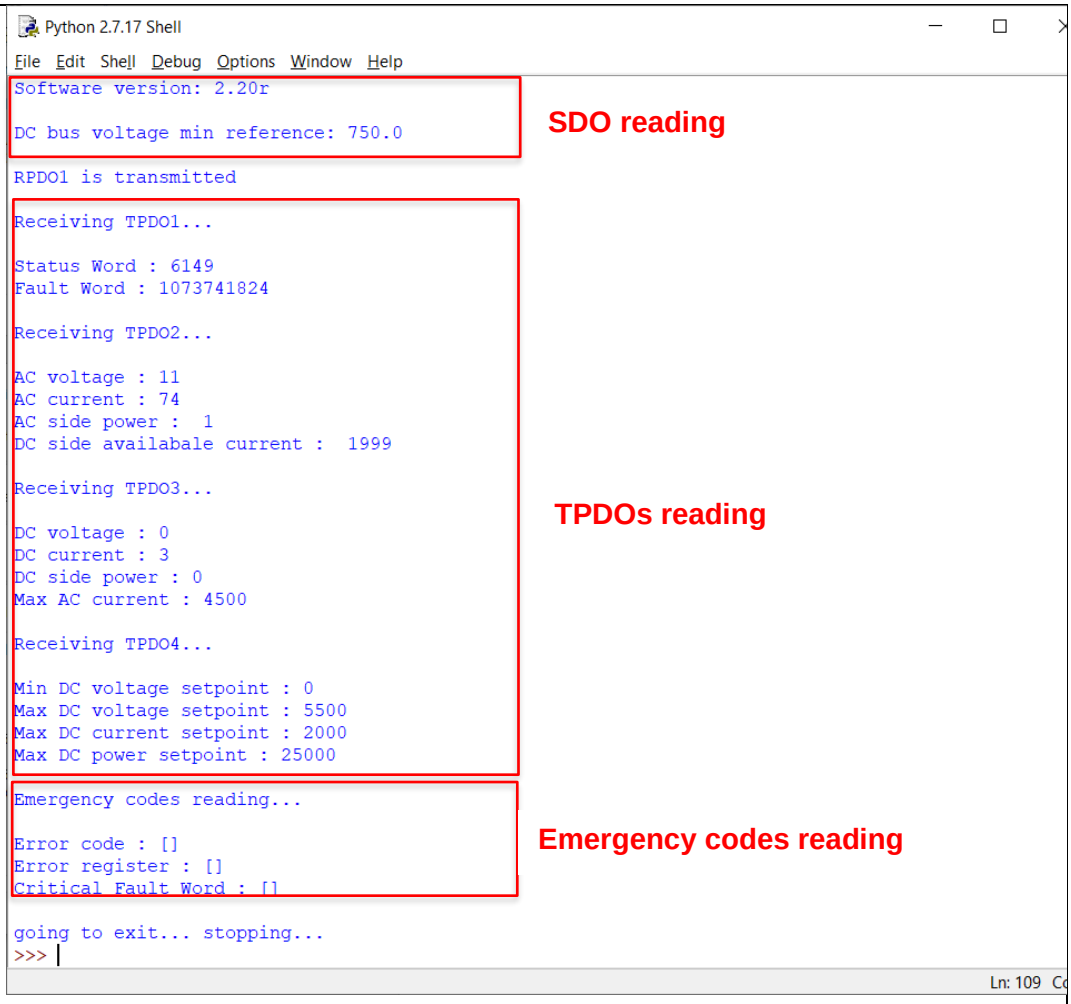
	# RPDO[1] message definition
	
	node.rpdo[1]['setPoints.state_Request'].raw = 5;
	node.rpdo[1]['setPoints.dcdc_currentOutSP'].raw = 6300;
	node.rpdo[1]['setPoints.dcdc_voltageOutSP'].raw = 4000;
	node.rpdo[1]['setPoints.pfc_iGridMaxSP'].raw = 4500;

	▪ Be careful with setpoint unities (refer to 3.2.8). In the following example, dcdc_currentOutSP (DC output current) is set to 6300 which corresponds to 6300 of 10 mA → 6300 x 0.01A=63 A, then the requested value is 63 A. ▪ state_Request is set to 5 for start charging (refer to 3.2.7). • Start sending RPDO0 periodically (every 0.1 s in this example)
	# Start RPDO[1] node.rpdo[1].start(0.1) **print** 'RPDO1 is transmitted', '\n'
Step 5 Reading TPDOs	• Reading TPDO0s. The following example is given for TPDO0 and the apporach can be reiterated for other TPDOs.
	# Read values from TPDO[1] node.tpdo[1].wait_for_reception() **print** 'Receiving TPDO1...', '\n' StatusWord = node.tpdo[1]['measurements.state_Current'].raw **print** 'Status Word :', StatusWord FaultWord = node.tpdo[1]['measurements.faultWord'].raw **print** 'Fault Word :', FaultWord, '\n'
	▪ TPDOs are configured to be transmitted after every nth sync message. The numberof sync message to be received before transmission of each TPDO is defined by the Transmission Type parameter. ▪ Transmission type for each TPDO are defined in Table 1.

Step 6 **Emergency codes reading**	• Reading emergency codes and critical fault world
	# Read emergency codes **print** 'Emergency codes reading...', '\n' error_code = [emcy.code **for** emcy **in** node.emcy.active] **print** 'Error code :', error_code error_register = [emcy.register **for** emcy **in** node.emcy.active] **print** 'Error register :', error_register error_data = [emcy.data **for** emcy **in** node.emcy.active] **print** 'Critical Fault Word :', error_data, '\n'
Step 7 **SDO reading**	• To read any object in the OD (parameter, signal, measurement …etc), an expedited SDO is used. The target object can be either stated by its variable name or index. • If the object is a part of a category, its variable and category names must be stated or its index and sub-index (see below *pfc_VintMin* example).

	# Read variables using SDO
	software_version = node.sdo['Manufacturer software version'].raw
	print 'Software version:', software_version, '\n'
	
	VintMinRef =node.sdo['commandSaturation']['pfc_VintMin'].raw
	print 'DC bus voltage min reference:', VintMinRef, '\n'

Step 8 Testing	• Run Python script • Execution result is shown below



- **Status Word interpretation :** 6149 = 1100000000101

b16 … b31	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1

System Mode is Charging (System Mode=5) as requested by Status Request in RPDO0. Also, b11=1 and b12=1 means that PFC and DCDC PWMs are on.

- To read measurement from TPDOs, unit conversion is necessary. As an example, the returned value for DC side available current is 0x1999 (decimal 6553) and regarding its unit in Error! Reference source not found. (10 mA), its physical value is 6553 x0.01A = 65.53 A.

3.2 CAN messages monitoring

In order to check Python script test functioning, MPU-25 GUI could be used in spy mode. The connection of a second CAN (NI USB-8473) transceiver is necessary to communicate with the GUI.

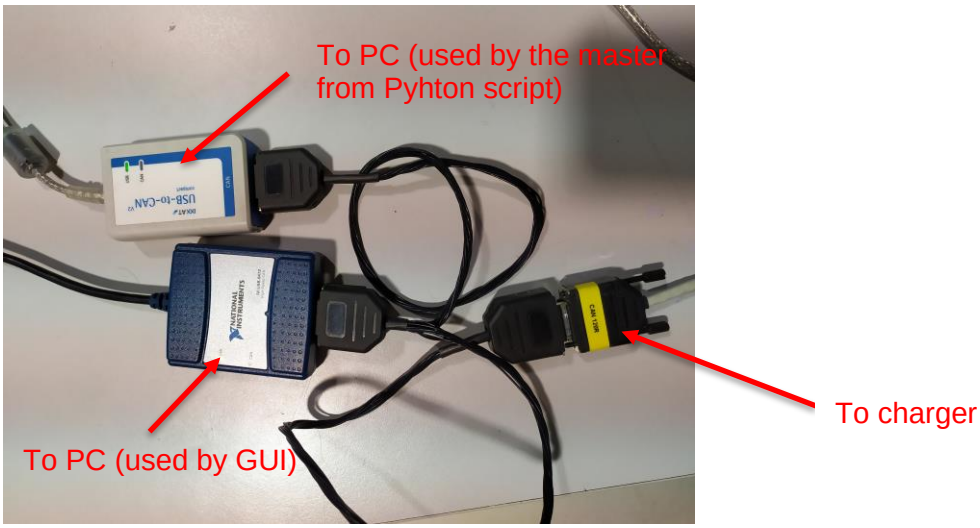


Figure 2 Transceiver configuration for CAN monitoring

Set-point autosend period and Slow Meas period must be set to 0 ms to cease transmission from GUI to charger (to be in spy mode) and slave address must be set to 80.

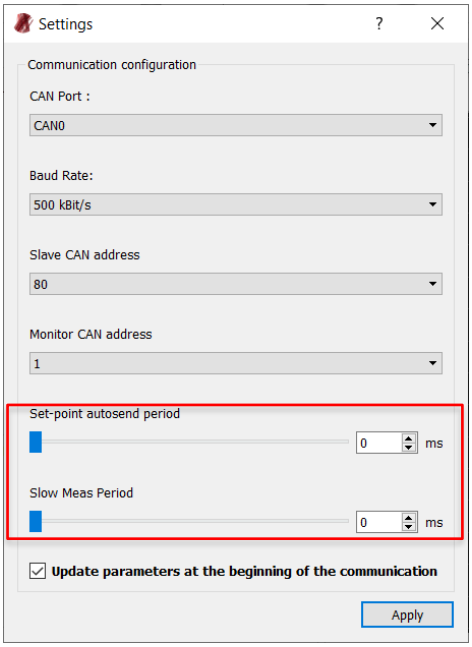


Figure 3 GUI settings

After Python script execution:

- Blue led of Charging mode must be lighted up on the charger and on the GUI.
- Green LEDs of PWM On DC/DC and PWM On PFC must be lighted up.
- The software version must be the same returned by Python script.

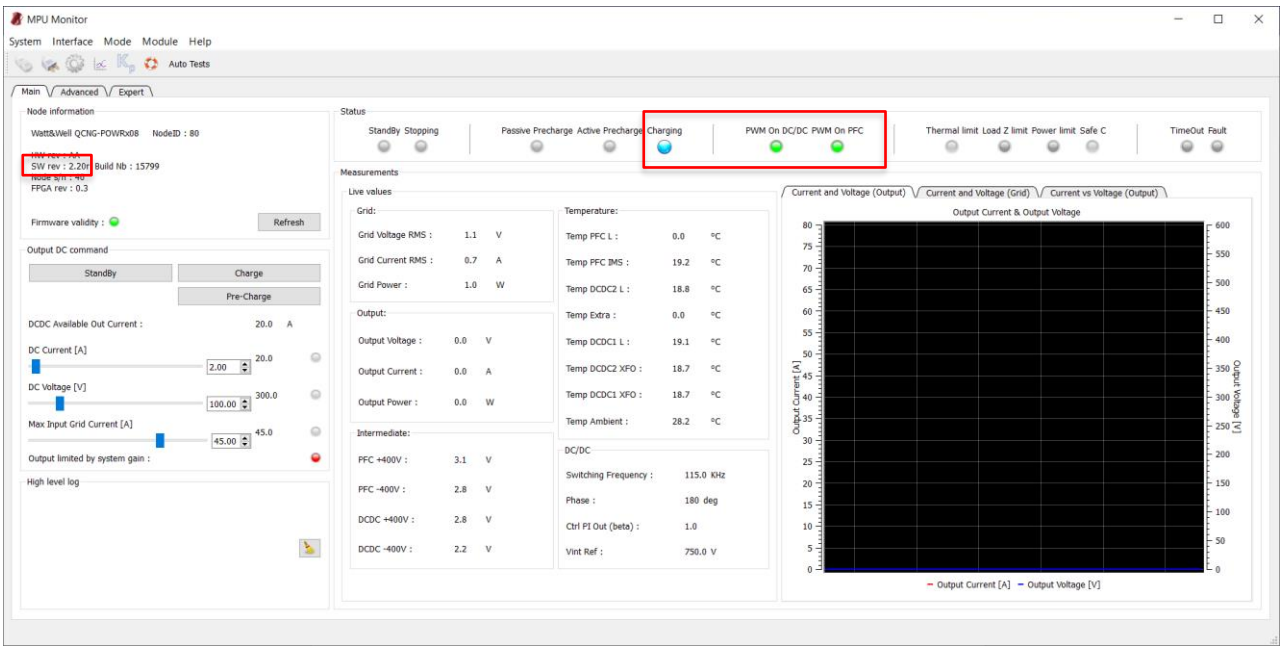


Figure 4 MPU-25 GUI in spy mode

- To check that RPDO0 message has been correctly received, it is possible to verify the received values of RPDO0 objects in the CANopen Device Manger (see Figure 5).

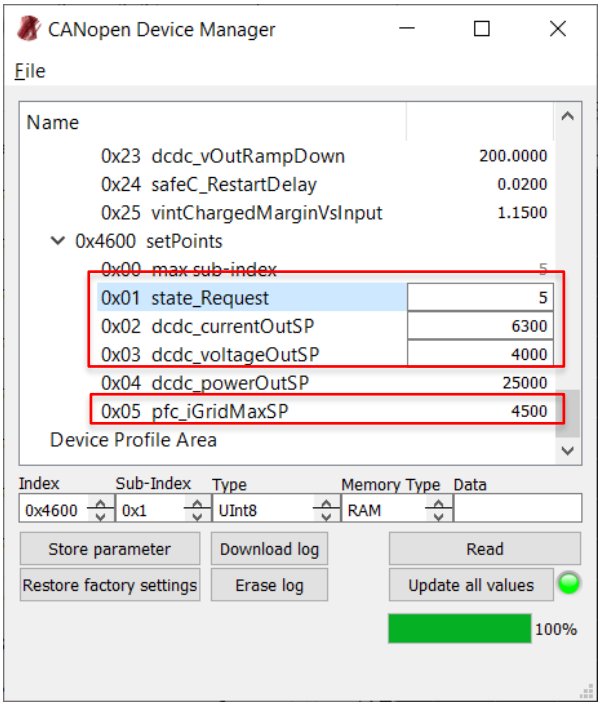
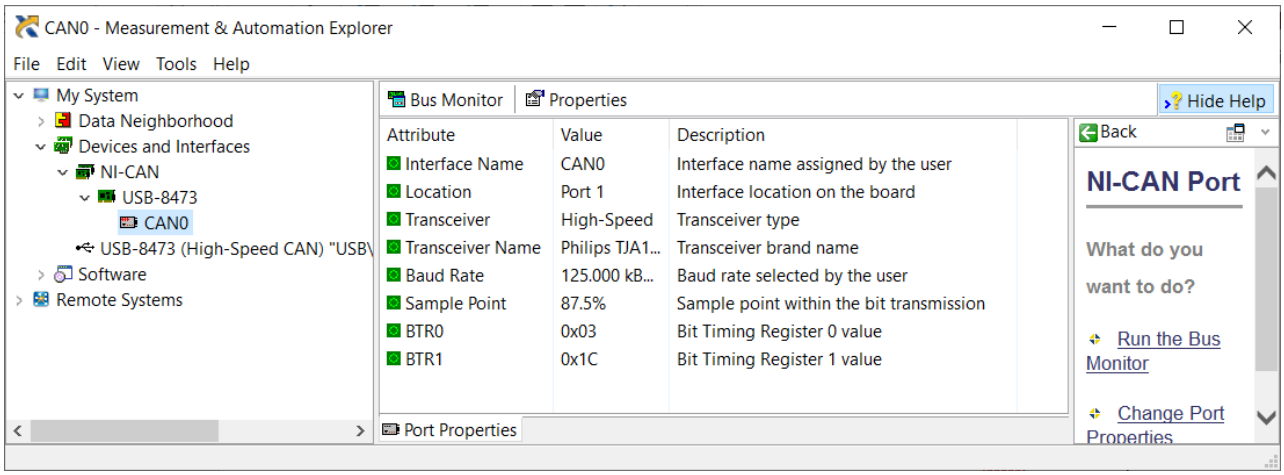


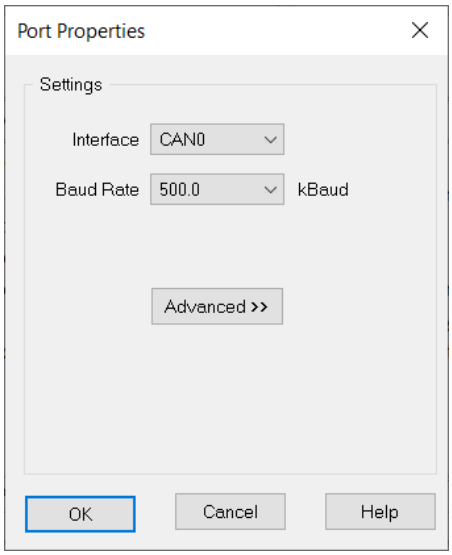
Figure 5 CANopen Device Manager

- Another way to spy on CAN messages is the utilization of NI Measurement and Automation Explorer (NI MAX). This software is included in NI-drivers package and installed automatically with the drivers. Configuration steps to monitor the CAN bus are as follows:

  - Start NI MAX
  - Go to *Devices and Interfaces* → NI CAN → USB-xxxx→ CAN0 (or select the corresponding interface)



  - In Properties, set the baud rate to 500.



- Click on Bus Monitor to start CAN bus monitoring. All frames generated by Python script must be visible.

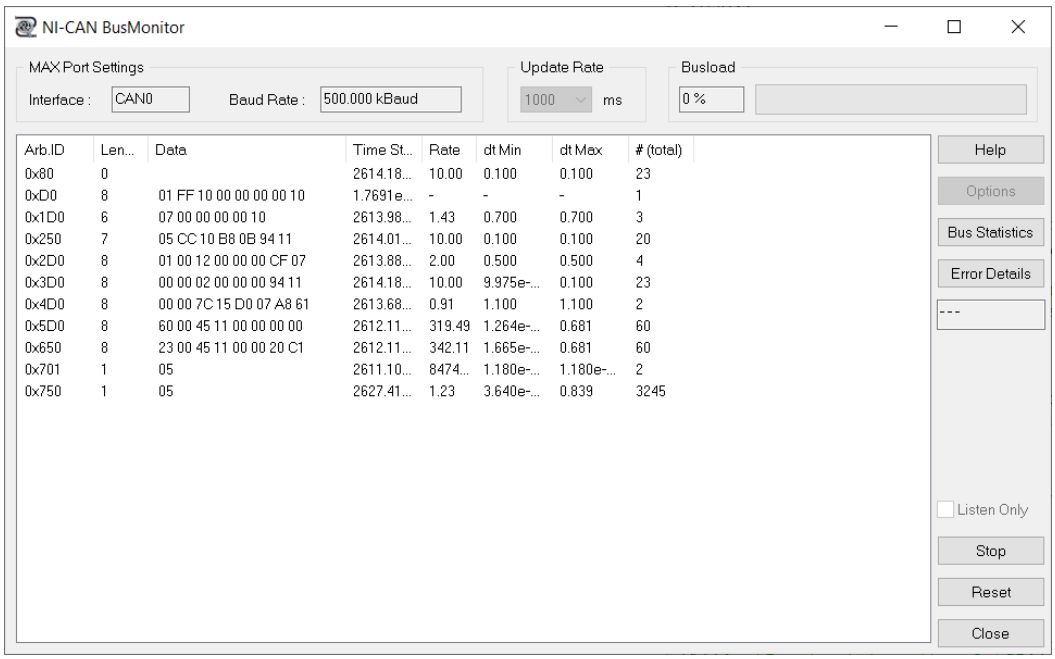


Figure 6 NI-CAN BusMonitor

If Sync message is sent every 0.1s, then TPDO1 with transmission type of 5 will be transmitted every 0.5 s (the value is comprised between *dt Min* and *dt Max*) which gives a rate of 2 (the message is transmitted twice per second).

3.3 Critical fault world interpretation

When critical fault occurs, Emergency message will return the Critical Fault Word as 5 bytes data (byte 3 to byte 7, keep in mind that byte 3 is not used).

To interpret the received data and determine which fault has occurred, bits must be compared to the fault word defined in 3.2.7.

The Critical Fault Word in the example below is decoded to illustrate how critical faults are determined. To create fault condition, AC power is disconnected

```
Emergency codes reading...

Error code : [65281]
Error register : [16]
Critical Fault Word : ['\x00\xc0\x01\x00\x00']
```

	Byte 7								Byte 6								Byte 5								Byte 4							
	C0								01								00								00							
Bits	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	23	22	21	20	19	18	17	16	31	30	29	28	27	26	25	24
	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Order of bytes and bits is due to CANopen endianness. Refer to section 4 for more details.

Then, occurred faults are determined by bits 6, 7 and 8 which corresponds to faults UV_PhaseAVoltageRMS, UV_PhaseBVoltageRMS and UV_PhaseCVoltageRMS. It corresponds to the created fault where there is no voltage at charger AC input.

4 Endianness

All numerical data types consisting of multiple bytes are transferred in CANopen (whether in SDO or PDO) in the Little-Endian format. Bytes are ordered by significance and the lower significant bytes come first. It means that last byte of binary representation of the multibyte datatype is stored first.

For example, the 32-bit hexadecimal number « 0xCDE11C0A» will be transmitted in CAN bus as follows

0A	1C	E1	CD