Arduino NMEA2000 library
31.12.2016

Kave Oy
Timo Lappalainen

SISÄLLYSLUETTELO

1 Thanks

At first I thank Kees Verruijt for his excellent CanBoat project. I originally started to learn NMEA 2000 via OpenSkipper project, which development I continued. PGN library on OpenSkipper and so on this library is mostly based on Kees work. Also it appeared that some code parts are probably originally from CanBoat even I have found them from other sources. Actisense message format handling I have learned from OpenSkipper project.

I also thank for anybody who has extended library with new PGNs or processor support.

2 Warning

First of all - as normal - you can use library but there is not any guarantee and you use it with your own risk!

3 Introduction

Arduino NMEA2000 library should make it simple to develop own NMEA2000 based devices. I created it because I wanted to get rid of limitations of expensive devices on market. With my first own device on my yacht I could replace three devices – NMEA0183 combiner, NMEA0183->NMEA2000 Converter, NMEA2000->PC converter - with single Arduino Due board and few extra chips.

To use NMEA2000 library you need basic skills for programming. I also try to write simple instructions to start from scratch. Even I have been doing sw and hw development for years, it took some time to dig information from internet to get started with Arduino and do simple tasks like installing library.

For using NMEA2000 library I prefer Arduino Due with MCP2562 chip. For simple things Arduino Mega with CANBUS shield or schematics found under https://github.com/ttlappalainen/NMEA2000/tree/master/Documents is ok. I have not tested smaller boards, but at least some of them should work.

4 Start from scratch and make your first NMEA2000 device

Lets think you have temperature sensor and you like to have it visible on NMEA2000 bus. Here I expect that:

- you already have Arduino Programming IDE installed. If not, do first “5.1 Getting started with Arduino Programming IDE”.
- you know how to import .zip libraries to Arduino. If not, read help on “5.2.1 Importing library from zip”.

- you have your Arduino board connected to USB on PC.
- NOTE! Arduino IDE serial monitor must be closed!
- You have right board selected from IDE Tools-Board" and right COM port selected from IDE Tools-Port:

4.1 Preparing to use NMEA2000 library

At first we have to download and install necessary libraries for Arduino and other useful tools.

- Download and install Arduino libraries:
  - https://github.com/ttlappalainen/NMEA2000/archive/master.zip
  - For Arduino Mega or other board with MCP2515 CAN bus controller
    https://github.com/ttlappalainen/NMEA2000_mcp/archive/master.zip
    https://github.com/ttlappalainen/CAN_BUS_Shield/archive/master.zip
  - For Arduino Due
    https://github.com/ttlappalainen/NMEA2000_due/archive/master.zip
    https://github.com/collin80/due_can/archive/master.zip
  - For Teensy 3.2
    https://github.com/ttlappalainen/NMEA2000_teensy/archive/master.zip
    https://github.com/sarfata/NMEA2000_teensy
  - For Atmel AVR processors internal CAN controller
    https://github.com/thomasonw/NMEA2000_avr
    https://github.com/thomasonw/avr_can

- Download and install "NMEA Reader"-application from http://www.actisense.com/products/actisense-software/nmeareader-eblreader/nmea-reader-downloads

4.2 Try NMEA2000 library with TemperatureMonitor example

4.2.1 Try on PC only

Now we are ready to try sample without N2k bus connection. So you need only Arduino board and PC. No CANBUS shield or extra chips yet!

- Open Example TemperatureMonitor sketch, which came with NMEA2000 library. Select from IDE "Open…"-libraries\NMEA2000-master\Example\TemperatureMonitor\TemperatureMonitor.ino

- uncomment lines
  // Serial.begin(115200);
  // NMEA2000.SetForwardStream(&Serial);
  and comment line
  NMEA2000.EnableForward(false);
  within setup() function.

- If you are using Arduino Mega, uncomment line
  // NMEA2000.SetDebugMode(tNMEA2000::dm_Actisense);
  within setup() function.
  We need to do this, since Mega board does not have build in CAN bus controller, so the library would fail, when it tries to open CAN on non debug mode.

- Upload sketch to board.

- Start “NMEA Reader” and from toolbar dropdown select your Arduino COM port.

Now you should see "ISO address claim" (PGN 60928), "Environmental parameters" (PGN 130310)), "Environmental parameters" (PGN 130311) and "Temperature" (PGN 130312) messages on “NMEA Reader”.

NOTE! If you want to upload sketch again to the board, you have to release Arduino programming COM port by selecting port “- NOT SELECTED –“ on “NMEA Reader”.

4.2.2 A bit deeper look in to the TemperatureMonitor example

First note that if you want to modify and save this sample, you have to save it on other location.

4.2.2.1 Change to use read real data

If you are happy with sending only cabin and water temperatures to the N2k bus, you simply change functions double ReadCabinTemp() and double ReadWaterTemp(). So e.g. if you have sensor providing 10 mV/ °C, you can change the function:

```
double ReadCabinTemp() {
  return (5.0 * analogRead(A0)*100/1024);
}
```

Do the same for ReadWaterTemp() (naturally you have to use other analog input e.g. A1) and you have your own ready NMEA2000 temperature monitor.

4.2.2.2 Remove unnecessary messages

With the library N2k messages will be sent by call NMEA2000.SendMsg(N2kMsg); Before sending you have to setup the variable N2kMsg. In that example it has been done for Temperature message with call:

SetN2kTemperature(N2kMsg, 1, 1, N2kts_MainCabinTemperature, ReadCabinTemp());

So if you do not need cabin temperature message (PGN 130312), simply delete that call SetN2kTemperature(…) and next call NMEA2000.SendMsg(N2kMsg); Similarly if you do not provide water temperature, delete calls SetN2kOutsideEnvironmentalParameters(…) and call NMEA2000.SendMsg(N2kMsg); after that. Now just upload sketch and open right port to “NMEA Reader” and you should see only "ISO address claim" (PGN 60928) and "Environmental parameters" (PGN 130311) on “NMEA Reader”.

4.2.2.3 Change the device information

On NMEA2000 bus each device will tell to the bus what it is. This can be found on sample under setup() function.

Call SetDeviceInformation is important for right functionality on the bus. If you have only this own device on the bus, you can leave it like it is. In other case take deeper look in to the explanation for the function.Note that if you add several own devices to the bus, you have to configure parameters for this function so that each device will get unic "name" as a combination of these parameters.

Call SetProductInformation is not so important, but defines nice to know information. If you have some "Multi Function Display" (MFD) on the bus, it will show on the device list information set by this function. So you can e.g. set the "Manufacturer's Model ID" to "Jack's temperature monitor" and version code and model version as you like, so on your MFD you can see that and choose right device.

4.2.3 Connecting your sample device to the NMEA2000 bus

So now we have sketch, which works on the PC. If your sketch runs and shows information in "NMEA Reader", it can be easily connected to the N2k bus. For electrical connection you need either CANBUS shield for Arduino Mega or you can build your own shield. You can find basic drawing for Arduino Mega on https://github.com/ttlappalainen/NMEA2000/blob/master/Documents/ArduinoMega_CAN_with_MCP2515_MCP2551.pdf and for Arduino Due on https://github.com/ttlappalainen/NMEA2000/blob/master/Documents/ArduinoDUE_CAN_with_MCP2562.pdf. There is also drawing for Teensy on https://github.com/ttlappalainen/NMEA2000/blob/master/Examples/TeensyActisenseListenerSender/Documents/Teensy_Actisense_listener_sender_schematics.pdf, which has additional USB with Pololu chip, which is only required for two USB solutions. So

- Prepare physical connection to the NMEA2000 bus
- comment line
  NMEA2000.SetDebugMode(tNMEA2000::dm_Actisense);
  within setup() function, if you have previously uncommented it.
- Upload the sketch
- On your NMEA2000 "Multi Function Display" (MFD, like Garmin GMI20) add temperature reading visible. For that I can not give more help, since each MFD has different user interface.

Now you should see the temperature reading on your MFD.

If you are not interested in N2k bus messages and you want to add your own debug code, you have to disable message forwarding by uncommenting line
NMEA2000.EnableForward(false);
which we commented on 4.2.1.

4.3 Try NMEA2000 library with WindMonitor example

This is exactly same as TemperatureMonitor, but sends wind data to the bus. Only necessary function names has been changed. So follow the “4.2 Try NMEA2000 library with TemperatureMonitor example”.

5 Arduino Programming IDE

5.1 Getting started with Arduino Programming IDE

- Download and install Arduino software for suitable OS from https://www.arduino.cc/en/Main/Software
- If you are using Teensy based boards, download and install also Teensyduino from https://www.pjrc.com/teensy/td_download.html. Note that with Teensyduino you have to check which version of Arduino IDE it supports!
- Start newly installed Arduino development IDE from “Arduino”-icon.
- Connect your board to USB
- Select right board for IDE from Tools-Board: NOTE! If you have “Arduino Due” do not mix it to “Arduino Duemilanove”. Select “Arduino Due (programming port)”. If you do not have that in selection list add support for due board selecting Tools-Board:-“Boards Manager…”. Then type “due” to filter box and click found “Arduino SAM Boards…”. Then on left click “Install”
- Select right COM port for IDE from Tools-Port: E.g. for Arduino Mega you should see on the list port: COMx (Arduino/Genuino Mega or Mega 2560)
- Now open your first sketch to IDE. Open File-Examples-01.Basics-AnalogReadSerial
- To see example output, open serial monitor: Tools-“Serial monitor”
- Upload sketch by IDE toolbar button or from menu Sketch-Upload.
- Now you should see board analog pin0 value running on serial monitor.

Now you are ready for your own developing.

5.2 Arduino Libraries

Arduino has both system wide and user libraries.

- System wide libraries are visible for all users and they are located as default e.g. on 64 bit Windows OS under "C:\Program Files (x86)\Arduino\libraries"
- User libraries are visible only for current user and they will be located as default e.g. on Windows OS under “Documents\Arduino\libraries" ("C:\Users\<user>\Documents\Arduino\libraries")

Arduino IDE installation automatically installs system wide libraries. When you add library with Arduino IDE, it will be added as default for you only. You can make some library visible for all users by moving installed library from your personal libraries to system wide libraries.

5.2.1 Importing library from zip

If someone points you direct link as .zip file to Arduino library, you need only:

- Download library to your PC
- On Arduino IDE select Sketch-"Include Library"-"Add .ZIP Library…"
- Navigate to you download path and select the .zip file you just downloaded and press Open.

So now you should have that library on your personal libraries.

5.2.2 Importing libraries from GitHub

Importing library from GitHub is also very simple. So you have heard or found good link to someone GitHub page.

- Open browser to the link. E.g. https://github.com/ttlappalainen
- Select repository, which should contain the Arduino library you are looking for. E.g. NMEA2000 on link above.
- Somewhere on page – currently on right – there is button "Clone or download". Click that and then on popup button "Download ZIP" and choose "Save to disk" and press "OK"
- Now just follow "5.2.1 Importing library from zip".

6 NMEA2000 library reference

6.1 Introduction

NMEA2000 is closed standard. You can buy documents for it for high price. As far as I have understood, then you also accept that you are not allowed to tell any third party about document contents. So all the information and experience I have has been collected from free documents and mostly from OpenSkipper project, which development I have been continued. So there may be errors on library. Anyway have tried to make it simple to use, but still giving "NMEA2000 compatibility".

6.2 Classes

6.2.1 Abstract base class tNMEA2000

The tNMEA2000 class contains functions for NMEA2000 handling like e.g. sending and reading messages. This is pure abstract class and to use real device you have to use suitable inherited class.

Normally you use functions within this class.

6.2.2 Inherited classes

There are own inherited class for each CAN interface type. As described below e.g. for Arduino Due you use due_can and NMEA2000_due libraries. Since Arduino IDE 1.6.6 it has been possible to include libraries within other headers, so to use any of currently tested board you can simply add includes

```
#include <Arduino.h>
#include <NMEA2000_CAN.h>
```

So the #include <NMEA2000_CAN.h> will then automatically select right library according to the board you have selected on IDE. If you want strictly control includes or the board will not be right selected by NMEA2000_CAN.h, use includes described on each inherited class.

6.2.3 Inherited class tNMEA2000_mcp

The tNMEA2000_mcp is class for using library with CANBUS shield or according to schematic https://github.com/ttlappalainen/NMEA2000/blob/master/Documents/ArduinoMega_CAN_with_MCP2515_MCP2551.pdf, which I have made for sample.

To use this class, you need to include in your project beginning:

```
#include <SPI.h>
#include <mcp_can.h>
#include <NMEA2000_mcp.h>
#define N2k_SPI_CS_PIN 53  // Pin for SPI Can Select
tNMEA2000_mcp NMEA2000(N2k_SPI_CS_PIN);
```

You can find the mcp_can library from https://github.com/ttlappalainen/CAN_BUS_Shield/archive/master.zip. Originally it was developed by SeedStudio https://github.com/Seeed-Studio/CAN_BUS_Shield, but they have not yet updated all features, what peppeve or I had added or fixed.

6.2.4 Inherited class tNMEA2000_due

The tNMEA2000_due class is for using library with Arduino Due, which has internal can bus controller. I personally prefer this board, since it is more powerful. It also has separate USB port. Also the physical interface to the bus is simpler. You can find sample schematic on https://github.com/ttlappalainen/NMEA2000/blob/master/Documents/ArduinoDUE_CAN_with_MCP2562.pdf.

To use this class, you need to include in your project beginning:

```
#include <due_can.h>
#include <NMEA2000_due.h>
tNMEA2000_due NMEA2000;
```

You can find the due_can library from https://github.com/collin80/due_can.

6.2.5 Inherited class tNMEA2000_teensy

The tNMEA2000_teensy class is for using library with Teensy 3.2 board, which has internal can bus controller. Physical interface is similar as with Arduino Due.

To use this class, you need to include in your project beginning:

```
#include <NMEA2000_teensy.h>
tNMEA2000_teensy NMEA2000;
```

6.2.6 Inherited class tNMEA2000_avr

The tNMEA2000_teensy clas is for using library avr based boards. I have not yet experience of them, but thomasonw uses them.

To use this class, you need to include in your project beginning:

```
#include <NMEA2000_avr.h>
tNMEA2000_avr NMEA2000;
```

6.2.7 Message container class tN2kMsg

This is the class for containing N2k (PGN) message. It contains pure data and functions to handle it. At least for now there is no automated system, which could you easily fill or read data for specific PGN. You have to know the PGN information fields like for NMEA0183 data fields. I and other developers have written functions, which has been defined on N2kMessages.h, which helps you the fill or read data on tN2kMsg object. As in example TemperatureMonitor you send PGN 130312 (Temperature) message with simple code:

```
tN2kMsg N2kMsg;

  SetN2kTemperature(N2kMsg, 1, 1, N2kts_MainCabinTemperature, ReadCabinTemp());
  NMEA2000.SendMsg(N2kMsg);
```

So as you see, you do not need to know much about tN2kMsg to use it. It is simple container. Note that there are functions SetN2k130312 and inline function SetN2kTemperature, which simply calls SetN2kPGN130312. This makes code easier to read (I hope), since you do not need to see PGN numbers there. But it's your choice, which function you will use.

I prefer that if you need something, which is not already on N2kMessages.h, I'll add it there or if you are familiar with NMEA 2000 message structures and Github, you can fork library and add it yourself. So then it is available for anybody.

6.3 Using tNMEA2000

There are several examples for using library with different functions like for only listen bus data (ActisenseListener.ino), sending temperature (TemperatureMonitos.ino) or wind data (WindMonitor.ino) from your own sources or displaying bus data somehow (DisplayData.ino and DisplayData2.ino). With combination of those or adding NMEA0183 library functions, you can have full control for your N2k bus information.

I have already own device, which reads data from old NMEA0183 devices and forwards it to the N2k bus, but same device also forwards data from bus to the PC. It also listens on input pin – MOB and

when that is activated, it will start to send route information back to the activation position. I have also temperature and fuel consumption monitors to the N2k bus.

So below is short description of member functions, which hopefully gives you better knowledge why something has been used on samples.

6.3.1 Device modes

In principle all devices should act as node on N2k bus. But if you are only reading messages on bus, why to tell anybody. So I have defined different modes how tNMEA2000 class behaves.

6.3.1.1 tNMEA2000::N2km_ListenOnly

This is default mode. The device simply listens data from bus and forwards it to the forward stream. Look example ActisenseListener, you need only 20 line for making device to read data on N2k bus.

Also if you like to make device, which shows some data on bus on e.g. TFT display, you can use this mode. There is simple example DataDisplay for that.

6.3.1.2 tNMEA2000::N2km_NodeOnly

In this mode device will only send data to the N2k bus. I also automatically informs itself to other devices on the bus and does required address claiming automatically. Device does not send as default anything to any forward stream.

Use this mode for device, which simply e.g. reads data from analog or digital input or NMEA0183 bus and sends it to the N2k bus. Look example TemperatureMonitor.

6.3.1.3 tNMEA2000::N2km_ListenAndNode

In this mode device works fully to both directions. It also forwards all data from bus to the forward stream, which you can define with function 6.3.4.18 SetForwardStream.

6.3.1.4 tNMEA2000::N2km_SendOnly

In this mode it is like tNMEA2000::N2km_NodeOnly, but it does not do automatic address claiming and does not forward any messages from N2k bus to stream. So this is useful, if you e.g. have some control pair like autopilot keypad/"control unit" and you want to fool that keypad sends something to the "control unit". Then you also need to know also source addresses of keypad and "control unit".

I have used this mode e.g. with example ActisenseSender and my NMEA Simulator found on http://www.kave.fi/Apps/ .

6.3.1.5 tNMEA2000::N2km_ListenAndSend

This is like the tNMEA2000::N2km_SendOnly mode, but it also forwards messages from N2k bus to the stream. In this mode one can have invisible gateway device between computer and N2k bus. This mode can be used e.g. if one has PC application, which is capable to read and send messages in Actisense format to serial port.

I have use this mode with example TeensyActisenseListenerSender and Actisense NMEA Reader and NMEA Simulator.

6.3.2 Message forwarding

Normally on N2k bus device either shows data from bus (MFD devices) or sends data to the bus (wind, GPS, temperature etc.). With Arduino you also get messages forwarded to the stream. In listen mode, device will read all messages from N2k bus and forwards all messages to the ForwardStream. As default it has been set to &Serial, so you only need to open it with Serial.begin(115200);

Messages will be forwarded as default in Actisense format. This is supported by at least some PC chart plotter applications. With default format Actisence "NMEA Reader", which we used on sample, can show message data. Better visualizer is OpenSkipper, on which you can tailor your own displays.

Note also that as default own messages send with tNMEA2000.SendMsg will be forwarded even you device has been set to node only mode. This may disturb your developing, if you e.g. want to write own clear text messages within your code. You can disable that by either

- tNMEA2000::EnableForward(false) to disable forwarding totally
- tNMEA2000::SetForwardOwnMessages(false) to disable own messages. But then, if your device is in listen mode, it will still forward messages from bus.
- tNMEA2000::SetForwardOnlyKnownMessages(true) to define that only known messages will be forwarded. The known messages are system messages and listed single frame or fast packet messages. See also 6.3.4.7 SetSingleFrameMessages, 6.3.4.8 SetFastPacketMessages, 6.3.4.9 ExtendSingleFrameMessages and 6.3.4.10 ExtendFastPacketMessages.

6.3.3 Debug mode

Debug mode is as default set to tNMEA2000::dm_None. In other debug modes device will not send anything to the N2k bus.

Debug mode tNMEA2000::dm_Actisense is usefull, if you do not have board with internal CAN bus controller, because in that mode it does not try to open CAN bus during sending message. Instead it simply writes message to the forward stream, which you can read e.g. with "NMEA Reader" and see that your device is sending message right.

Debug mode tNMEA2000::dm_ClearText defines that messages will be send to forward stream as clear text. Unfortunately on clear text there are only source, destination, priority data length and PGN shown as clear text. Data itself will be shown as hex.

6.3.4 Member functions

6.3.4.1 SetN2kCANMsgBufSize

With this function you can set size of buffer, where system stores incoming messages. The default size is 5 messages.

Some messages are just single frame messages and they will be read in and handled immediately on call to ParseMessages. For multi framed fast packet messages there is no guarantee that all frames will arrive in order. So these messages will be buffered and saved until all frames has been recived.

If it is not critical to handle all fast packet messages like with N2km_NodeOnly, you can set buffer size smaller like 3 or 2 by calling this before open.

6.3.4.2 SetN2kCANSendFrameBufSize

With this function you can set size of buffer, where system saves frames of messages to be sent. The default size is 40 frames so totally 320 bytes.

When sending long messages like ProductInformation or GNSS data, there may not be enough low level buffers for sending data successfully. This depends of your hw and device source. Device source has effect due to priority of getting sending slot on low level device. If your data is critical, use buffer size, which is large enough.

E.g. Product information takes totally 134 bytes. This needs 20 frames. GNSS contains 47 bytes, which needs 7 frames. If you want to be sure that both will be sent on any situation, you need at least 27 frame buffer size.

6.3.4.3 SetProductInformation

If you are using device modes tNMEA2000::N2km_NodeOnly or tNMEA2000::N2km_ListenAndNode, it would be good that you set this information. With this you can set how e.g. Multi Function Displays (MFD) will show your device information. This is not critical, but nice to have it right.

See example TemperatureMonitor.ino.

6.3.4.4 SetDeviceInformation

If you are using device modes tNMEA2000::N2km_NodeOnly or tNMEA2000::N2km_ListenAndNode, it is critical that you set this information. Note that you should set information so that it is unique over the world! Well if you are making device only for your own yacht N2k bus, it is enough to be unique there. So e.g. if you have two temperature monitors made by this library, you have to set at least first parameter UniqueNumber different for both of them.

Device information will be used to choose right address for your device (also called node) on the bus. Each device must have own address. Library will do this automatically, so it is enough that you call this function on setup to define your device.

6.3.4.5 SetConfigurationInformation

With this function you can set configuration information, which will be saved on device RAM. As an alternative you can set configuration information saved on progmem with 6.3.4.6 SetProgmemConfigurationInformation

Configuration information is just some extra information about device and manufacturer. Some MFDs shows it, some does not. E.g. NMEA Reader can show configuration information.

6.3.4.6 SetProgmemConfigurationInformation

With this function you can set configuration information, which will be saved on device program memory. See example BatteryMonitor.ino.

As default system has build in configuration information on progmem. If you do not want to have configuration information at all, you can disable it by calling SetProgmemConfigurationInformation(0);

6.3.4.7 SetSingleFrameMessages

As a default library has a list of known messages. With this function user can override default list of single frame messages. See also 6.3.4.9 ExtendSingleFrameMessages.

6.3.4.8 SetFastPacketMessages

As a default library has a list of known messages. With this function user can override default list of fast packet messages. See also 6.3.4.10 ExtendFastPacketMessages.

Note that if incoming fast packet message is not known, it will be treated as single frame message. So if you want to handle unknown fast packet message, you need to duplicate frame collection logic from library to your code. So it is easier to have fast packet messages listed on library, if you want to handle them.

6.3.4.9 ExtendSingleFrameMessages

As a default library has a list of known messages. With this function user can add own list of known single frame messages. Note that currently subsequent calls will override previously set list.

6.3.4.10 ExtendFastPacketMessages

As a default library has a list of known messages. With this function user can add own list of known fast packet messages. Note that currently subsequent calls will override previously set list.

Note that if incoming fast packet message is not known, it will be treated as single frame message. So if you want to handle unknown fast packet message, you need to duplicate frame collection logic from library to your code. So it is easier to have fast packet messages listed on library, if you want to handle them.

6.3.4.11 ExtendTransmitMessages

System should respond to PGN 126464 request with messages system transmits and receives. Library will automatically respond with system messages it uses. With this method you can add messages, which your own code sends. Note that this is valid only for device modes 6.3.1.2 tNMEA2000::N2km_NodeOnly and 6.3.1.3 tNMEA2000::N2km_ListenAndNode.

6.3.4.12 ExtendReceiveMessages

Method is like 6.3.4.11 ExtendTransmitMessages, but extends messages you code handles.

6.3.4.13 SendIsoAddressClaim

This is automatically used by class. You only need to use this, if you want to write your own behavior for address claiming.

6.3.4.14 SendProductInformation

This is automatically used by class. You only need to use this, if you want to write your own behavior for providing product information.

6.3.4.15 SendConfigurationInformation

This is automatically used by class. You only need to use this, if you want to write your own behavior for providing configuration information.

6.3.4.16 SetMode

With SetMode you can define how your node acts on N2k bus. See 6.3.1 Device modes.

Note that with this function you can also set default address for your device. It is preferable that once your device has been connected to the bus, it tries always use last used address. Due to address claiming, your device may change its address, when you add new devices to the bus. So it would be good to save last used address to the EEPROM and on startup read it there and use it as parameter for SetMode. You can check has your address you set originally by SetMode changed by using function 6.3.4.25 ReadResetAddressChanged and you can read current address by function 6.3.4.24 GetN2kSource.

So you could do next:

```
void setup() {
...
NMEA2000.SetMode(tNMEA2000::N2km_NodeOnly,GetLastSavedN2kAddressFromEEPROM());
...
}

...
void loop() {
  SendN2kTemperature();
  NMEA2000.ParseMessages();
  if (NMEA2000.ReadResetAddressChanged() ) {
    SaveN2kAddressToEEPROM(GetN2kSource());
  }
}
```

See example TemperatureMonitor.ino.

6.3.4.17 SetForwardType

With this function user can set how messages will be forwarded to the stream. Possible values are:

- fwdt_Actisense (default), forwards messages is Actisense format. Some navigation softwares can read this format.
- fwdt_Text, forwards messages to output port in clear text. I see this useful only for testing with normal serial monitors.

6.3.4.18 SetForwardStream

As default, forward stream has been set to null. For e.g. Arduino Due you can set it to SerialUSB, so you can use Serial for other things. You can of coarse use any stream available on your device.

See example ActisenseListenerSender.ino.

6.3.4.19 Open

You can call this on Setup(). It will be called anyway automatically by first call of ParseMessages().

6.3.4.20 SendMsg

When you want to send some message to the N2k bus, you call this. Before calling you have to prepare tN2kMsg type of message e.g. by using some function in N2kMessages.

Note also that as default tNMEA2000 object is as default in tNMEA2000::N2km_ListenOnly mode. So if you want to send messages, you have to set right mode in Setup().

Function returns true, if message was sent successfully, otherwise it return false. SendMsg may fail, if there is not room for message frames on sending buffer or device is not open.

SendMsg does not always send message immediately. If lower level sending function fails, SendMsg will buffer message frames and try to send them on next call to SendMsg or ParseMessages. So to have reliable sending, you need sending buffer, which is large enough.

See example TemperatureMonitor.ino.

6.3.4.21 ParseMessages

You have to call this periodically on loop(), otherwise tNMEA2000 object will not work at all. Note also that it is not good practice to have any delay() on you loop(), since then also handling of this will be delayed.

See example TemperatureMonitor.ino.

6.3.4.22 SetMsgHandler

If you want to do something with messages read from N2k bus, easiest is to set message handler, which will be then called by ParseMessages, if there are new messages. This is the case e.g. if you have LCD display on your Arduino and you want to show some fluid level on it.

See example DataDisplay.ino or DataDisplay2.ino

6.3.4.23 SetISORqstHandler

Devices on N2k bus may request from your device can it handle requested PGN. If you want to respond for ISO request, you should set this handler. The handler will be called by ParseMessages, if there is ISO request.

Note that when you send request message with SendMsg and it fails, it is your responsibility to take care of sending response again later. If your sending buffer is larger enough, it is very uncommon that SendMsg fails.

6.3.4.24 GetN2kSource

With this function you can get you device current address on the N2k bus.

See 6.3.4.16 SetMode and 6.3.4.25 ReadResetAddressChanged

6.3.4.25 ReadResetAddressChanged

With this function you can check has your device address you initiated with SetMode been changed after last call. It will be good to call this time to time and if address has been changed, save new address to the EEPROM.

See 6.3.4.16 SetMode and 6.3.4.24 GetN2kSource

6.3.4.26 EnableForward

Set true as default. With this you can control will bus messages be forwarded to forward stream. See 6.3.2 Message forwarding.

6.3.4.27 SetForwardSystemMessages

Set true as default. With this you can control will system messages like address caliming, device information be forwarded to forward stream.

If you set this false, system messages will not be forwarded to the stream.

6.3.4.28 SetForwardOnlyKnownMessages

Set false as default. With this you can control will unknown messages be forwarded to forward stream.

If you set this true, all unknown message will not be forwarded to the stream. Note that this does not effect for own messages.

Known messages are listed on library. See 6.3.4.7 SetSingleFrameMessages

, 6.3.4.8 SetFastPacketMessages, 6.3.4.9 ExtendSingleFrameMessages and 6.3.4.10 ExtendFastPacketMessages.

6.3.4.29 SetForwardOwnMessages

Set true as default. With this you can control will messages your device sends to bus be forwarded to forward stream.

6.3.4.30 SetHandleOnlyKnownMessages

Set false as default. With this you can control will unknown messages be handled at all.

Known messages are listed on library. See 6.3.4.7 SetSingleFrameMessages, 6.3.4.8 SetFastPacketMessages, 6.3.4.9 ExtendSingleFrameMessages and 6.3.4.10 ExtendFastPacketMessages.

6.3.4.31 SetDebugMode

If you do not have physical N2k bus connection and you like to test your board without even CAN controller, you can use this function. See 6.3.3 Debug mode.