

Yacht Devices

User Manual

Yacht Devices NMEA 2000 Bridge YDNB-07
also covers models
YDNB-07N, YDNB-07R

Firmware version
1.42

2022

© 2022 Yacht Devices Ltd. Document YDNB07-005. May 24, 2022. Web: <http://www.yachtd.com/>

NMEA 2000® is a registered trademark of the National Marine Electronics Association. SeaTalk NG is a registered trademark of Raymarine UK Limited. Garmin® is a registered trademark of Garmin Ltd.

Contents

Introduction	4
Warranty and Technical Support	6
I. Product Specification	7
II. Installation and Connection to NMEA 2000 or CAN Networks	9
III. LED Signals	11
IV. MicroSD Slot and Card's Compatibility	12
V. Loading of Programs into the Device	14
VI. Structure and Basic Syntax of the Program	15
VII. Description of the Settings	17
VIII. Programming the Device	21
IX. Optimization and Performance	34
X. Debug and logging functionality	38
XI. Firmware Updates	40
XII. Program protection and encryption	41
Appendix A. Troubleshooting	42
Appendix B. List of NMEA 2000 Messages of the Device	44
Appendix C. Device Connectors	45

Package Contents

Device	1 pc.
This Manual	1 pc.
Stickers for MicroSD slot sealing	6 pc.

Introduction

Yacht Devices NMEA Bridge unifies two physical NMEA 2000 networks into a single logical network, smoothly exchanging messages between them. The Device also allows to organize arbitrary filtering, processing and routing of any CAN bus messages.

This can accomplish the following tasks:

1. **Bypass the physical limits of NMEA 2000 networks** concerning length of networks (100 meters for regular cable and 250 meters for heavy or mid-type cable) and concerning the maximum number (50) of physical devices attached to the network. On a network with address capacity of 252, multiple bridges can be engaged to expand to around 250 physical devices.
2. **Isolate devices from each other.** Using the simple filter, you can block transmission of all or of selected messages from a given device in a separate subnet.
3. **Ensure proper functioning of equipment.** Correct the transducer offset of the depth sounder, or “delete” invalid data in messages from equipment that is only partially operational using a 2- or 3-line script.
4. **To ensure compatibility of equipment** from different generations. You can create and send any type of NMEA 2000 message using data from other messages in the network.
5. **Diagnose malfunctions in the NMEA 2000 network.** The Device can record network messages and debug data from custom programs on a MicroSD card in a text file. You can view the data in a standard text editor on a smartphone or tablet with a MicroSD slot, there is no need for a computer. You can even create and edit programs for the Device right on your phone!
6. **Create custom gateways** that do not meet NMEA 2000 standards. One of the CAN-interfaces on the Device has high-voltage galvanic isolation and can operate at a higher supply voltage.
7. **Create custom gateways** joining two CAN networks with arbitrary protocols. Both 11-bit (CAN A) and 29-bit (CAN B) frames are supported. One of the network can operate at any baud rate from 50 kbps to 1000 kbps. Though device programming language was not designed for full-fledged CAN applications, one can create, for example, a gateway between CAN-based Propulsion System, Battery Charger or Servo to NMEA 2000.

8. **Create custom NMEA 2000 data processing device**, for example, implement complex digital switching rules for Yacht Devices NMEA 2000 Digital Switching equipment, such as: delayed and timed switching, auto-off, conditional switching or trigger warnings on a chartplotter when certain data values are over the threshold.

Programming the Device requires knowledge of NMEA 2000. Copy of NMEA 2000 standard can be purchased from the National Marine Electronics Association: <http://www.nmea.org>.



Yacht Devices would like to note that a NMEA 2000 network might contain important devices such as a deep sounder, magnetic compass and autopilot. Failure or incorrect operation of these devices can result in serious accidents and fatalities. When programming the Device, you must be fully aware of all the implications. Before making a sea-going trial, conduct mandatory training for the vessel's crew.

Warranty and Technical Support

1. The Device warranty is valid for two years from the purchase date. If the Device was purchased in a retail store, when applying under a warranty case, the sale receipt may be requested.
2. The Device warranty is terminated in case this Manual instructions violating, case integrity breach, repair or modification of the Device without manufacturer's written permission.
3. If a warranty request is accepted, the defective Device must be sent to the manufacturer.
4. The warranty liabilities include repair and replacement of the goods and do not include the cost of equipment installation and configuration, as well as shipping the defective Device to the manufacturer.
5. Manufacturer responsibility in case of any damage as a consequence of the Device operation or installation is limited to the Device cost.
6. The manufacturer is not responsible for any errors and inaccuracies in guides and instructions of other companies.
7. The Device requires no maintenance. The Device's case is non-dismountable.
8. If the event of a failure, please refer to Appendix A. before contacting the technical support.
9. The manufacturer accepts applications under the warranty and provides technical support only via e-mail or from authorized dealers.
10. Manufacturer contact details and a list of the authorized dealers are published on the website: <http://www.yachtd.com/>.

I. Product Specification

NMEA 2000 network connector, CAN 2
(SeaTalk NG, see note below)

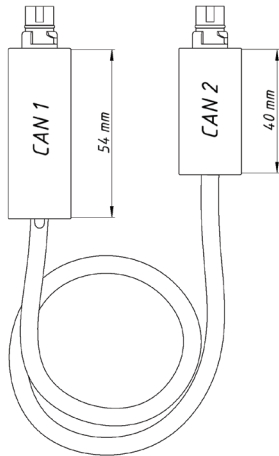
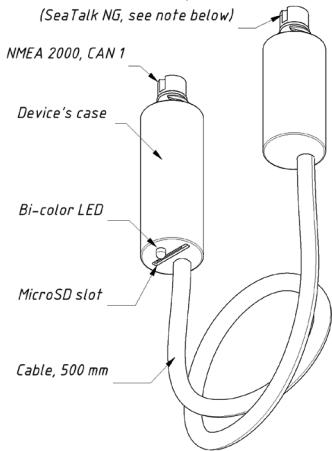


Figure 1. Drawing of YDNB-07R model of the Bridge

Our devices are supplied with different types of NMEA 2000 connectors. Models with the suffix R at the end of model name are equipped with NMEA 2000 connectors compatible with Raymarine SeaTalk NG (as at the picture above). Models with the suffix N are equipped with NMEA 2000 Micro Male connectors (see Appendix C).

Device parameter	Value	Unit
Power supply voltage, CAN1 interface	9..16	V
Average current consumption, CAN1	38	mA
Load equivalency number, CAN1	1	LEN
Power supply voltage, CAN2 interface	9..30	V
Average current consumption, CAN2	13	mA
Load equivalency number, CAN2	1	LEN
Galvanic isolation electrical strength between CAN1 and CAN2	2500	V _{RMS}
Reverse polarity protection on both CAN1 and CAN2 interfaces	Yes	—
Operating temperature range	-20..55	°C
Cable length	500	mm
Device's case length (without connector) CAN1 / CAN2	54/40	mm
Weight without MicroSD card	52	g



Yacht Devices Ltd declares that this product is compliant with the essential requirements of EMC directive 2004/108/EC.



Dispose of this product in accordance with the WEEE Directive. Do not mix electronic disposal with domestic or industrial refuse.

II. Installation and Connection to NMEA 2000 or CAN Networks



Never connect both connectors of the Device to the same NMEA 2000 or CAN network. This can flood the network with infinite forwarding of messages and cause a temporary inoperability of the network.

The Device requires no maintenance. When deciding where to install the Device, choose a dry mounting location. Avoid places where the Device can be flooded with water; this can damage it.

Device can be directly plugged into to the network backbone connector. YDNG-07N model can also be connected via standard "DeviceNet NMEA 2000 Micro" drop cable. If you have a CAN bus with a non-standard physical layer, connect CAN bus rails and power rails according to Device pinout given in Appendix C. Before connecting the Device, turn off the bus power supply. Refer to the manufacturer's documentation if you have any questions regarding the use of connectors:

- SeaTalk NG Reference Manual (81300-1) for Raymarine networks
- Technical Reference for Garmin NMEA 2000 Products (190-00891-00) for Garmin networks

After connecting the Device, close the lock on the connection to ensure water resistance and reliability.

The microcontroller of the Device is powered by the CAN1 interface. The Device will not work until the CAN1 interface is powered up. If you want to start up the Device for familiarization purposes or you implement a custom data processing device which does not need to send data between CAN1 and CAN2 interfaces by design, CAN2 interface can be left unconnected.

The Device has a LED which flashes red or green. When Device CAN1 interface is powered, LED will produce a series of 2 flashes 5 seconds apart. If this does not happen, see Appendix A.

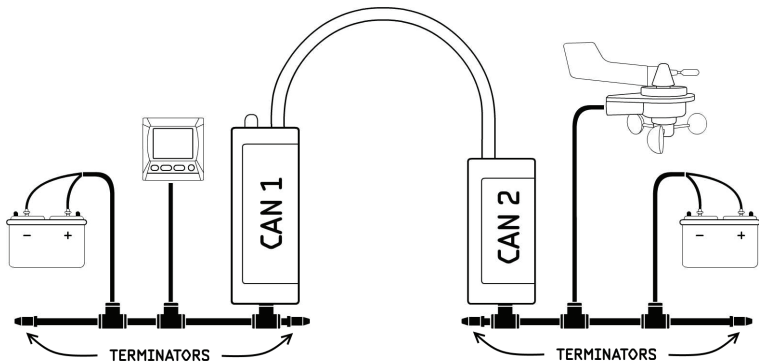


Figure 2. Typical NMEA 2000 bridging application network layout

Please remember that a NMEA 2000 network requires a separate power supply and two terminators, on each backbone end. If the Device is used as a bridge, connecting two existing NMEA 2000 network segments, you have to make sure to add terminators to each backbone segment and power both segments from a 12V power supply. More information on this topic is available in the above listed documents from Raymarine and Garmin.

III. LED Signals

1. **Signal with period of 5 seconds, two flashes of the LED.** The first flash indicates the condition of the CAN1 interface network. Green LED flash indicates that data has been received or successfully sent on CAN1 interface within the last 5 seconds period, red if not. The second flash indicates the condition of the CAN2 interface network.

The Device can be configured to receive only a limited set of NMEA 2000 messages (see Section VII.3), the remaining messages are filtered at the hardware level. In this regard, some NMEA 2000 networks can indicate a red light much of the time, even when the network is functioning normally. In this case, to check the connection, turn one device that is on the network (e.g. the chart plotter) off and on again. The status of the network will be displayed with green flashes for some time as the device is powering up and connecting.

2. **Three flashes (colors may vary), one time after inserting the MicroSD card into the Device.** See Section V.
3. **Long flash (3-second), red or green.** Diagnostics mode started / finished, see Section X.
4. **Five green flashes when NMEA 2000 network is turned on.** The Device has the MicroSD inserted with a firmware update, the firmware is updated (see Section XI).

IV. MicroSD Slot and Card's Compatibility

The Device has a slot for a MicroSD card that allows you to configure the Device (see Section V) and update the firmware (see Section XI).



The Device slot has a 'push-push' mechanism that works on a spring and ensures proper card fixation. Improper loading or unloading (withdrawing your finger too quickly or not waiting for the click) can result in the card being propelled out of the Device up to 5 meters. To avoid possible eye injury, loss of or damage to the card, and other hazards, insert and remove the card with caution.

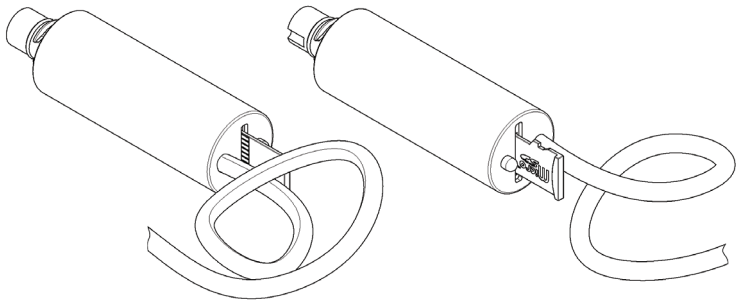


Figure 1. Device with MicroSD card (contact pads side visible at left, label side at right)

Since the MicroSD slot is usually not in use when the Device is working, we recommend to seal it with the sticker supplied with the Device or with a piece of tape to prevent water from entering the Device through the slot.

The Device supports MicroSD memory cards of all capacities and classes. The MicroSD card must be formatted on a personal computer before it can be used with the Device. The Device supports the following file systems: FAT (FAT12, FAT16, MS-DOS) and FAT32. It does not support exFAT, NTFS, or any other file systems.

Be careful when inserting the MicroSD into the Device. The card is inserted with the label side towards the LED and with the contact pads side towards the cable.

V. Loading of Programs into the Device

Place YDNB.CFG file with the program into the root directory of a MicroSD card with a FAT or FAT32 file system. Insert the MicroSD card into the Device. After a few seconds, you should observe three LED flashes:

1. **Three red flashes** indicates that the card cannot be read.
2. **A green followed by two red flashes** indicates that the YDNB.CFG file cannot be found on the memory card and the current Device configuration was saved to the YDNBSAVE.CFG file.
3. **A red followed by a green and another red flash** indicates that the YDNB.CFG file contains errors and was not accepted by the Device. The text file YDNBERR.TXT was created in the root directory of the memory card, comprising an error log.
4. **Three green flashes** indicates that the file has successfully been loaded into the Device. The text file YDNBSAVE.CFG was created in the root directory of the card, comprising the current program and used settings.

The Device performs compilation of the program text into bytecode. Before the Device saves a program in the YDNBSAVE.CFG file, it decompiles the bytecode to text. This is why the contents of the YDNB.CFG and YDNBSAVE.CFG files can differ from each other.

The YDNB.CFG file must contain at least one interpretable line of code (a setting, filter etc.) without errors, in order to be loaded into the Device.

To modify the current program, insert a memory card into the Device that does not contain a YDNB.CFG file. The LED of the Device will flash green, red, red. This indicates that a YDNBSAVE.CFG file, comprising the current program, was saved onto the card. This program can be modified, saved as YDNB.CFG, and loaded back into the Device.

VI. Structure and Basic Syntax of the Program

The program defines algorithms and rules for the processing and forwarding of NMEA 2000 or other CAN messages that the Device receives via the CAN1 and CAN2 interfaces.

The program consists of settings, built-in functions, filter subprograms and comments. Settings, functions and filters are described in detail in the later sections of this Manual.

Comments in the program are added after the # symbol. Comments can be situated at the beginning of a line as well as after interpretable program text.

Settings can be set anywhere in the program, except for inside subprograms of filters. Nevertheless, we recommend declaring settings before filters.

Example 1.

```
# Example N1
FW_CAN1_TO_CAN2=ON           # Allow forwarding of all mismatched messages
FW_CAN2_TO_CAN1=OFF         # Setting for other direction
match(CAN2,0x01FD0600,0x01FFFF00) # 1st filter
{
    # Empty subprogram, matched messages will be dropped
}
match(CAN2,0x00000010,0x000000FF) # 2nd filter
{
    send(CAN1)                # Forward of matched message to CAN1 interface
}
match(CAN1,0x00000020,0x000000FF) # 3rd filter (for CAN1)
{
    # No send(), matched messages will be dropped
}
# End of program
```

A filter consists of a header, which begins with the keyword `match()` with parameters defining the incoming CAN frames "match criteria" followed by a data processing subprogram in specially created programming language.

The order of the filters is significant. Device will match received messages against the filters in an exact same order they are specified in the program. After a successful match, the message will be processed by corresponding filter subprogram and will not be processed by further filters.

In case there is no match with any filters, the Device follows the forwarding settings. If message forwarding is enabled for the interface (by default, yes), the given message will be sent to the other CAN interface. If disabled, the message will be discarded.

Looking ahead, we explain the operation of this program.

The program in *example 1* above has only two forwarding settings and three filters.

First, we have two default forwarding rules – those will be applied to messages for which no corresponding `match()` subprogram was defined. `FW_CAN1_TO_CAN2=ON` instructs the Device to forward all messages received on CAN1 interface to CAN2 interface. `FW_CAN2_TO_CAN1=OFF` instructs the Device to not forward (drop) all messages received on CAN2 interface to CAN1 interface.

1st filter will intercept PGN 0x1FD06 received on CAN2 interface – from any device address, however, as its subprogram is empty, this PGN will be dropped.

2nd filter will intercept any PGN received on CAN2 interface – but only from device with address 0x10 – and as the subprogram dictates, it will send the message to CAN 1 interface. As PGN 0x1FD06 was already processed by the 1st filter, forwarding is disabled via `FW_CAN2_TO_CAN1=OFF` and there are no more filters for CAN2, the net result will be that all PGNs from CAN2 device 0x10 will be forwarded to CAN2, except 0x1FD06.

3rd filter will intercept all PGNs received on CAN1 interface from device with address 0x20 and block it. As we have CAN1 -> CAN2 forwarding enabled via `FW_CAN1_TO_CAN2=ON`, the net result will be that all PGNs from CAN1 will be forwarded to CAN2, except those from device 0x20.

VII. Description of the Settings

Note that a vertical bar (pipe) is used in the descriptions below to separate alternative setting values. ON|OFF means, that the setting can have two different values — ON or OFF.

1. Forwarding of messages

```
FW_CAN1_TO_CAN2=ON|OFF  
FW_CAN2_TO_CAN1=ON|OFF
```

Sets the default forwarding rules — defines the behaviour for messages with no corresponding `match()` filters. ON — forward, OFF — do not forward (drop).

2. Assembly of NMEA 2000 fast-packet messages

```
PGNS_TO_ASSEMBLY=x
```

x — from one to five PGN, entered as decimal or hexadecimal values, separated by commas.

NMEA 2000 Standard defines a special protocol extension for PGNs with payload overall length higher than 8 bytes (from 9 to 223 bytes) — payload is transmitted in a series of standard CAN frames, each having a payload length from 8 to 1 byte (refer NMEA 2000 Standard, section "3.1 Fast-packet messages").

The Device can "assemble" NMEA 2000 fast-packet PGNs from the series of CAN frames and present the whole PGN payload to the program. Messages not completely assembled will be discarded. Assembly takes a lot of RAM though, so you can select only up to five PGNs for assembly.

Note that assembly also takes time, as Device will wait until the last CAN frame will be received before passing the assembled PGN to the program, this will introduce a delay. Use frame-by-frame data processing technique whenever possible (see Section IX).

NMEA 2000 Standard does not require a strict order of fast-packet frames on the bus. Device CAN controller has 3 "outboxes", thus fast-packet CAN frames can be put in different "outboxes" in the outgoing queue, this may lead to fast-packet frames to be sent out-of-order. Some NMEA 2000 equipment can not handle this situation well, in such cases use the following data flow control setting:

```
STRICT_QUEUE=OFF|CAN1|CAN2|ON
```

This setting forces strict order of *outgoing* fast-packet frames for CAN1, CAN2 or both interfaces (ON). By default is disabled (OFF). Enabling has a slight performance penalty.

3. Hardware filters

```
CAN1_HARDWARE_FILTER_y=f,m
```

```
CAN2_HARDWARE_FILTER_y=f,m
```

y – number of the hardware filter, decimal number from 1 to 7;
f – filter value, decimal or hexadecimal number (29 significant bits);
m – filter mask, decimal or hexadecimal number (29 significant bits).

The Device can filter messages received from the CAN1 and CAN2 interface at hardware level, which in some cases can reduce the load on the microprocessor by a factor of up to a 100. The message selection is carried out through the 29-bit identifier of CAN frame (CAN ID), which contains the message priority, PGN, the sender's address and (in some cases) the recipient's address.

Formal match comparison expression for both hardware and software filters is:

```
if ( ( CAN_FRAME_ID AND mask ) == filter ) then MATCH else NO_MATCH
```

Messages are only passed to the program if they match one of the hardware filters. It is possible to set up to 7 custom hardware filters for each interface, with numbers from 1 to 7.

Device also has a system hardware filter number 0 which can not be modified by user:

```
CAN1_HARDWARE_FILTER_0=0x00E80000, 0x01F90000
```

```
CAN2_HARDWARE_FILTER_0=0x00E80000, 0x01F90000
```

The filter (first parameter) sets bits for the comparison with the message identifier and the mask (second parameter) indicates the bits whose comparison result is significant. (1 – corresponding by position bit of message CAN ID should match the bit set in filter exactly, 0 – can be any).

Thus, the system hardware filter passes only messages with the following PGNs: 0xE800 (ISO Acknowledgement), 0xEA00 (ISO Request), 0xEC00 (ISO Transport protocol), 0xEE00 (ISO Address Claim).

If no custom hardware filter for the interface is defined in the program, a filter that accept all messages will be automatically added for the given interface:

```
CAN1_HARDWARE_FILTER_1=0,0
```

So if no custom hardware filter is set in the program, all messages will be passed to the program.

4. NMEA 2000 instance

```
DEVICE_INSTANCE=x  
SYSTEM_INSTANCE=y
```

x – number from 0 to 255, y - number from 0 to 15.

These settings allow you to set the Device's NMEA 2000 "Device Instance" and "System Instance" values which are used for generation of NMEA 2000 "Device Information" (NAME) transmitted in the ISO Address Claim message.

The default value of both settings is 0. These settings will not be saved to the YDNBSAVE.CFG file if they have the default values.

Note that when CAN2 baud rate is set to 250 (CAN2_SPEED=250), setting SYSTEM_INSTANCE=8 (or higher) prevents sending device own "Address Claim" PGN from CAN2

5. Message slots initialization

SLOTx=aabbccdd... or SLOTx=aa bb cc dd...

x – slot number, 1 – 3;

aabbccdd – sequence of bytes (1 – 229), hexadecimal values.

Device has 3 buffers, called "SLOTS" which can be used as outgoing CAN frames templates or as a data storage buffers. SLOTx settings allow to fill slots buffers with predefined data before program start. Buffer format is described in VIII.2. You can load slot contents to main message buffer (DATA) with `load()` function and overwrite slot content with `save()` function (see VIII.8)

Example of non-addressed ISO Request of ISO Address Claim:

```
SLOT1= 00FFEA18 FF 03 00EE00
```

Here 00FFEA18 is CAN ID for "ISO Address Claim" in LSB-first byte order (message priority + PGN number + device address = 18EAFF00) FF – single frame PGN marker, 03 – payload length in bytes, 00EE00 – PGN payload.

6. Speed of the CAN2 interface

```
CAN2_SPEED=x
```

x - speed in kbps, factory setting is 250, allowed values are 50, 125, 250, 500 and 1000.

CAN2 interface speed can be set within a range from 50 to 1000 kbps (NMEA 2000 has 250 kbps speed). This allows interoperability with various CAN networks. CAN1 interface speed cannot be changed, it has fixed speed of 250 kbps.

VIII. Programming the Device

We have created a special programming language for the Device similar to common scripting languages. There are number of limitations imposed by tasks and computing capabilities of the microcontroller.

Program consists of ASCII text, each line is either a statement, comment or blank line. Each new statement must start on a new line. "Line breaks" are not supported. There is no special "statement terminator" symbol.

Programming language supports integers and rational numbers, positive and negative. Decimal separator for rationals is dot. Hex numbers should be prefixed with 0x (x in lower case).

It is possible to use 7-bit ASCII symbols in single quotes (like 'a', '1'), they will be interpreted as corresponding ASCII symbol number (ex.: 'a' == 97).

Reserved identifiers like constants (DATA, M_PI), interface identifiers (CAN1, CAN2), data type identifiers (UINT32, FLOAT, etc.) must be entered in upper case.

Programming language supports only one user-defined function `func()`, also you can call built-in functions `get()`, `set()`, `cast()`, `send()`, `sin()` and others.

1. Defining filters

A filter is defined by the keyword `match()` with 3 parameters in parentheses and subprogram code in curly brackets. Subprogram may be empty:

```
match(interface_id, filter, mask)
{
}
```

interface_id – the interface identifier: CAN1, CAN2 or ANY;
filter and mask – the numeric values of the filter and mask.

Formal match comparison expression for both hardware and software filters is:

```
if ( ( CAN_FRAME_ID AND mask ) == filter ) then MATCH else NO_MATCH
```

For example, this filter

```
match(ANY, 0x00000000, 0x01FFF800)
{
}
```

will match all 11-bit CAN frames (for all CAN IDs in range 0x000...0x7FF we will have a match — as all such IDs AND 0x01FFF800 expression yields 0)

The keyword `match()` is not translated into bytecode, so the comparison of the specified filters with the messages received is very quick.

The program may contain up to 20 filters.

2. Message buffer

The subprogram has access to the message and its 29-bit CAN identifier that is stored in the buffer described in Table 1.

Table 1. Message buffer structure

Offset	Length in bytes	Description
0	4	29-bit identifier CAN message (LSB first)
4	1	0xFF – indicator of a single-frame NMEA 2000 message (CAN message with 29-bit identifier); 0xFE - indicator of a CAN message with an 11-bit identifier; 0x00..0xE0 in steps of 0x20 – indicator of the NMEA 2000 fast-packet message (sequence counter); other values are reserved
5	1	Message length (1..223)
6	223	Message data

By modifying the values in the buffer, the subprogram can modify messages and even create new messages. Regardless of the actual length of the received message, the subprogram has access to all 229 bytes of the buffer.

Please note that if a NMEA 2000 fast-packet PGN is not included in the PGNS_TO_ASSEMBLY list, the program will be called for each CAN message in the sequence, and the byte at offset 4 will be set to 0xFF.

3. Sending a message

To send message stored in the buffer a built-in `send()` function is used:

```
send(interface_id)
```

interface_id — interface through which the message is sent, CAN1 or CAN2. This parameter can be omitted, in this case the message will be sent through the interface opposed to the one from which the processed message was received.

In order to avoid embarrassing mistakes with unpredictable consequences, we recommend always to use the `send()` without a parameter, wherever possible.

Note that the filter subprogram is responsible for forwarding the message being processed. If your subprogram does not contain a call to `send()`, then the processed message will not be forwarded.

4. Variables, operators, and expressions

There are 26 global variables available to the programmer, with names from A to Z. Variable name is not case-sensitive. Variables can be one of the following types that are defined implicitly and automatically converted to the necessary type when operations are performed.

- INT8 — signed 1-byte integer;
- UINT8 — unsigned 1-byte integer;
- INT16 — signed 2-byte integer;
- UINT16 — unsigned 2-byte integer;
- INT32 — signed 4-byte integer;

- `UINT32` – unsigned 4-byte integer;
- `FLOAT` – 4-byte floating point number (IEEE 754-1985 single precision).

When you turn on the Device, all variables are initialized to 0 type `INT32`.

The following arithmetic, logical and binary operations are supported (listed in order of priority):

- `*` (multiplication), `/` (division), `%` (the remainder of integer division)
- `+` (addition), `-` (subtraction), `<<` (binary shift to the left), `>>` (binary shift to the right), `|` (logical OR), `^` (XOR), `&` (logical AND)
- `=` (assignment)

Parentheses can be used to change the order of operations:

```
A = (3 + 5) * 2 # Equivalent A = 16
```

When you assign values to variables, the result type is automatically selected:

```
A = 1           # INT32
B = 2.0        # FLOAT
C = '1'       # 0x31, UINT8
D = B * C     # 0x31 * 2.0 = 49 * 2.0 = 98.0, FLOAT
E = 0xFF00AA3F # UINT32
```

Built-in `cast()` function allows to explicitly set an expression type:

```
[type] cast(expression, type)
```

Where *type* is one of the types listed above. For example:

```
B = cast(1.5, INT8)      # 1, INT8
C = cast(B << 2, FLOAT)  # 4.0, FLOAT
```

Note that the compiler does not optimize algorithms, try to perform calculations efficiently.

5. Modification of the message buffer

To modify a message in the buffer and read data from it, there are, respectively, two built-in functions `set()` and `get()`:

```
set(expression1,type,expression2)
[type] get(expression1, type)
```

expression1 — expression converted to `UINT8` type, offset of the first byte of data in the buffer;

type — the data type identifier (see VIII.4);

expression2 — an expression, the result of which is will be casted to the specified type and placed in a buffer with the offset that is specified in *expression1*.

Addressing of the data in the buffer starts at zero. For more convenient access to the data, we recommend using the built-in constant `DATA`, which points to a first byte of the message payload.

Example 2.

```
match(CAN1,0x1F50B00,0x1FFFF00)
{
    A = get(DATA+1, UINT32)    # get 4-byte integer from message
    set(DATA+1, UINT32,A + 20) # set the corrected value
    send()                    # send message to CAN2
}
```

In Example 2, a filter for Water Depth (PGN 0x1F50B) is implemented. The subprogram retrieves the value of the depth (four-byte integer value, bytes 1-4 of the incoming message payload data) and sets the value back into the buffer, adding 20. After that, the message is sent (to CAN2 interface). As a result, depth readings will be increased by 20 centimeters in CAN 2 network.

6. Flow control statements `if()` and `while()`

Conditional statement `if()` and loop statement `while()` support the following comparison operators: `>` (more than), `<` (less than), `==` (equal), `!=` (not equal), `<=` (less than or equal), `>=` (more than or equal). The *else* block is optional:

```

if (expression1 operator expression2) {
    # do stuff
}
else
{
    # do another stuff
}

```

Example 2 above does not take into account that depth transducer can send special depth values 0xFFFFFFFF "not available", 0xFFFFFFF0 "out of range" or 0xFFFFFFF1 "reserved". Using `if()` operator we will check if the data value is within allowed range:

Example 3.

```

match(CAN1,0x1F50B00,0x1FFFF00)
{
    A = get(DATA+1, UINT32)      # get depth value
                                # as 4-byte integer from buffer

    if (A < 0xFFFFFFF1-20)     # value + required 20 cm offset valid?
    {
        set(DATA+1, UINT32,A + 20) # add 20 cm offset
    }
    send()                      # send message to CAN2
}

```

Device supports flow control statement `while()` for creating pre-test loops:

```

while ( expression1 operator expression2 ){
    # do stuff
}

```

Comparison operators are the same as for `if()` statement. As `while()` can make an infinite loop, after 3 seconds the execution of the loop will be terminated and program execution will be continued from outside

the loop. If maximum execution time is reached, Device produces "FUNC Execution timeout (3000 ms)" message in the TEXT diagnostics log.

7. Real time functions

The majority of NMEA 2000 data items have a short life span. You need to track received data items timestamps to determine if data is still valid (e.g. in order not to use old values if data source device goes off the bus and/or stops sending data)

```
UINT32 timer()  
UINT32 timediff(expression)
```

The `timer()` function returns the number of milliseconds since the Device was turned on. The timer value overflows (resets back to zero) approximately every 49 days. The `timediff()` function returns the difference (in milliseconds) between the current timer value and timestamp value passed as an argument. Function can handle timer overflows and will return correct time difference value — if the difference is less than 49 days. See also: `heartbeat()`, VIII.11.

8. Storage buffers

```
save(slot)  
load(slot)
```

The `Save(slot)` function stores data from the message buffer into a specified SLOTx buffer. `Load(slot)` function overwrites current message buffer with data fetched from a specified SLOTx buffer. Three SLOTS identifiers are supported: SLOT1, SLOT2, SLOT3. Note: you may set SLOT contents at the start of the program (see VII.5).

9. Getting the Bridge address

```
UINT8 addr()
```

The `addr()` function returns current Bridge NMEA 2000 device address, it has the same value on both CAN1 and CAN2 interfaces.

Use when you need to change PGNs source device address and make them appear as if they were sent by the Bridge itself.

Example 4.

```
# Processing of Actual Pressure messages (PGN 0x1FD0A) and generation of the
# Environmental Parameters messages (PGN 0x1FD06)

match(CAN2,0x1FD0A00,0x1FFFF00)
{
    send()                # Forward the original message
    A = get(DATA + 2,  UINT8)    # Extract the data type
    if (A == 0)           # Is it atmospheric pressure?
    {
        B = get(DATA + 3,  INT32)    # Extract the pressure value
        if (B < 0x7FFFFFFD)    # Check the validity of the value
        {
            set(1,  UINT8,6)        # Change PGN from 0x1FD0A to 0x1FD06
            set(DATA + 1,  UINT32, 0xFFFFFFFF)    # Set the unused fields
            set(DATA + 5,  UINT16, B / 1000)    # Convert and set pressure
            send(CAN1)            # Send Environmental Parameters to CAN1
            set(0,  UINT8,  addr())    # Replace the address with the address
                                   # of the Bridge
            send(CAN2)            # Send Environmental Parameters to CAN2
        }
    }
}
```

Program in Example 4 receives "Actual Pressure" PGN 0x1FD0A from CAN2 interface and forwards it to CAN1 interface. Suppose we have a legacy device on CAN1 subnet which does not understand "Actual Pressure" PGN, so Bridge will generate an extra "Environmental Parameters" PGN 0x1FD06 for atmospheric pressure and will send it to CAN1 using its own device address. Original PGN 0x1FD0A will be send "as is", with no change in device address.

10. Program initialization

Like filters, the initialization section is defined by the keyword `init()` with no parameters, after which its subprogram follows in curly brackets:

```
init()
{
}
```

The `init()` function can be used to initialize program variables. Its subprogram code will be executed on Device power-up prior to any other code. It is not possible to `send()` data from `init()`, as `init()` code is executed before the Device finishes NMEA 2000 "address claim" procedures. All `send()` calls from `init()` will be ignored.

11. Heartbeat

Like filters or `init()`, `heartbeat` is defined by the keyword `heartbeat(ms)` after which its subprogram follows in curly brackets. `Heartbeat` requires integer parameter (constant) which specifies repeat interval in milliseconds.

```
heartbeat(1000)
{ # This code will be executed every second
}
```

Function `heartbeat()` can be used when a code must be executed periodically. The `heartbeat()` is first called from the user's program immediately after `init()`.

12. User function: `func()` and `call()`

Device supports one user function `func()` which can be called from any other subprogram via `call()`. Like filters or `init()`, user function is defined by the keyword `func()` after which its subprogram follows in curly brackets. Example 5 shows `func()` called from `heartbeat()` function via `call()`.

Example 5.

```
# Fibonacci Sequences Generator

DIAGNOSTICS=45 # record first 45 numbers

init(){

    # set two first numbers in Fibonacci sequence
    A=cast(0,UINT32) # F0
    B=cast(1,UINT32) # F1
}

# calculate next Fibonacci number and write it to log file
func(){

    C = A + B          # F(n) = F(n-1) + F(n-2)
    log(C)             # write F(n) to log file
}

# produce next number each second, so 45 in total
heartbeat(1000){

    call()             # calculate next Fibonacci number
    A = B              # store F(n-2) for the next round
    B = C              # store F(n-1) for the next round
}
```

Recursion — `call()` inside `func()` — is also allowed, but maximum recursion depth is limited to 10.

If maximum recursion level is reached, Device produces "Maximum recursion depth is reached" message in the TEXT diagnostics log.

13. Math functions and constants

Floating point constants:

`M_PI` = π = 3.14159...
`M_2PI` = $2 * \pi$
`M_PI2` = $\pi / 2$

High precision floating point constants:

`M_180P` = $180 / \pi$
`M_1800P` = `M_180P` / 10
`M_P180` = $\pi / 180$

Trigonometric functions:

`FLOAT sin(x)` `FLOAT cos(x)` `FLOAT tan(x)`

Where x argument is an angle in radians.

Inverse trigonometric functions:

`FLOAT asin(y)` `FLOAT acos(y)` `FLOAT atan(y)`

Returns an angle in radians. Note: feeding argument values outside of function domain yields zero.

Square root:

`FLOAT sqrt(z)`

Note: feeding negative value into `sqrt()` yields zero.

2-argument arctangent

`FLOAT atan2(x, y)`

Returns the angle on Euclidean plane, in radians, between the positive x axis and the ray to the point (x, y).

Note: `atan2(0, 0)` yields zero.

14. Software Reset

Call `reset()` function to reboot the Bridge, it can be used to restart the program and reset both CAN interfaces, for example, by reception of the particular CAN frame or by timer.

15. Passing parameters to program in real time

```
UINT8 param(slot)
```

Function `param()` can handle special "YD:PARAM" command entered into the Bridge's "Installation Description String 2" (PGN 126998) via NMEA 2000-to-PC gateway. This might come handy when you need to adjust some program parameters and observe response in real time, for example, when adjusting PID values on a CAN Servo.

"YD:PARAM" command format is:

```
YD:PARAM hexits_string
```

Where `hexits_string` is a payload — sequence of up to 60 hexadecimal digits (hexits) — characters in range [0–9, A–F]. This mechanism allows to send up to 30 arbitrary bytes for the Device to process.

If command is accepted Bridge will reply and add "OK" to the "Installation Description String 2". If there is an error in payload like invalid symbol or total hexits count exceeds 60, Device will reply with "YD:PARAM ERR".

You can enter hexits delimited by spaces or without them, with leading zero or not, in upper or in lower case. Device will remove all spaces in a reply and convert hexits to upper case, so any of the commands below:

```
YD:PARAM 5 6 7 1A f
YD:PARAM 05 06 07 1a 0F
YD:PARAM 0506071A0F
```

Will yield the same response and same SLOT contents:

```
YD:PARAM 0506071A0F OK
```

Note: if all 30 bytes are entered, the Device will not add "OK" to reply due to string size limit of "Installation Description String".

Function `param()` takes one of the SLOTS as argument and returns number of bytes read from the "YD:PARAM" command payload. Target SLOT will be filled with up to 30 data bytes from the command

payload. To get payload bytes just `load()` this SLOT to copy contents into work buffer. SLOT can hold up to 229 bytes but this configuration method allows you to set only the first 30 bytes; all other bytes in the SLOT will be cleared.

If no command was given, command payload is empty or command is incorrect, SLOT will not be modified and `param()` will return 0.

Example 6.

```
# Fuel level tank simulator. Sends PGN 127505 "Fluid Level".
# Allows to set arbitrary "fuel level" via "YD:PARAM yy" command,
# where yy - desired fuel level (hexadecimal value, in %)

# prepare PGN 127505 "Fluid Level" with 100% full diesel tank, fluid instance = 0
SLOT1 = 0111F219 FF 08 00A861FFFFFFFFFFFF

# main loop executed each second
heartbeat(1000)

{
    # new tank level settings received via "YD:PARAM" ?
    if (param(SLOT2) == 1)    # is only one byte read (correct payload length)?
    {
        load(SLOT2)          # copy slot to work buffer
        A = get(0, INT8) * 250 # get new fluid level %, convert to N2K resolution
        load(SLOT1)          # load PGN 127505 into work buffer
        set(DATA+1, INT16, A) # update "fluid level %" data field
        save(SLOT1)          # save SLOT1 with modified "fluid level %" data
    }
    # Send PGN 127505 "Fluid Level"
    load(SLOT1)
    send(CAN1)
}
```

IX. Optimization and Performance

The transfer time of a typical CAN single-frame with a full payload length of 8 bytes is about 520 microseconds. Device does not waste any CPU resources handling CAN bus data link layer communications, those are offloaded to CAN controller hardware.

Delay between message reception and transmission — for messages that do not match any filter — is within 100 microseconds — even if maximum number of filters (20) is used.

Of the 100 total microseconds, only 40 microseconds are used for the comparison with filters (program execution). The rest of the time is spent processing interrupts of the CAN controller, placing the received message into the program input queue, removing messages from the program output queue and then handing it over to the CAN controller to send.

The runtime (for messages matching a filter) of Example 2 is 240 microseconds, and of Example 9 — 515 microseconds. The total delays in the Device are 300 and 575 microseconds respectively.

Device has a software queue for 100 inbound and 100 outbound messages, with a maximum lifetime of messages in the queue of 50 milliseconds. The device's performance is sufficient for practical applications, even in highly loaded networks. As long as egregious programming errors are avoided, no overload issues should occur.

1. Use hardware filters

One of the most common Bridge use cases is data filtering between two network segments. Usage of hardware filters shown in Example 7 below is extremely effective.

Example 7.

```
FW_CAN1_TO_CAN2=ON
FW_CAN2_TO_CAN1=ON
CAN1_HARDWARE_FILTER_1=0x00000000, 0x00FFFFFF
match(CAN2,0x1FD0A00,0x1FFFF00)
{
    # some required processing here...
}
```

Thanks to the system hardware filters (see VII.3), messages which are required for the normal operation of the NMEA 2000 network (ISO Acknowledgement, ISO Request, ISO Address Claim) will be forwarded between CAN1 and CAN2 interfaces unmodified.

In this program a hardware filter for CAN2 interface is not defined, so Device will add a default filter 1 to accept all messages from CAN2 and pass them to filters for further processing (see VII.3). For CAN1 interface, CAN1_HARDWARE_FILTER_1 is set up to block all PGNs on CAN1 interface. As a result, all messages received on CAN1 will be dropped before they reach `match()` filters, which saves a lot of CPU resources.

2. Use `match()` instead of multiple `if()`

One may be tempted to use multiple `if()` statements to process different PGNs inside a single filter like in Example 8.a below.

This technique is inferior to `match()` filters usage, as `match()` filters use hardware-accelerated CAN ID comparison — adding additional `match()` filters does not introduce extra overhead or wasted CPU resources. On the other hand, `if()` is executed on Device virtual machine, hence is several times slower.

Example 8.a (incorrect)

```
match(CAN1,0x00000010,0x000000FF)      # sent from 0x10 address?
{
    p = (get(0,UINT32) >> 8) & 0x1FFFF # get the PGN
    if (p == 0x1FD0A)                  # is it "Actual Pressure" message?
    {
        # process data
    }
}
```

Example 8.b (correct)

```
match(CAN1,0x1FD0A10,0x1FFFFFF) # "Actual Pressure" PGN from 0x10 address?
{
    # process data
}
```

3. Use optimizations instead of relying on a compiler

Current version of Device compiler can not optimize expressions.

For example, in this code:

```
A = time()           # get device uptime in ms
B = A / ( 60 * 1000 ) # convert to minutes
C = A / ( 60 * 60 * 1000 ) # convert to hours
```

Multiplication will be executed 3 times, which can be easily avoided:

```
A = time()           # get device uptime in ms
B = A / 60000        # convert to minutes
C = A / 3600000      # convert to hours
```

If you have a lot of calculations in your program, try to optimize them.

Future updates of the software will place special focus on issues of optimization during compilation into bytecode.

4. Avoid fast-packet PGNs assembly whenever possible

In most cases, the data in the message can be modified without pre-assembly. Consider, for example, hacking the log distance (a criminal offense in most countries) using the Distance Log message (PGN 0x1F513).

Example 9.

```
PGNS_TO_ASSEMBLY=0x1F513
match(CAN1, 0x1F51300, 0x1FFFF00)
{
    set(DATA+6, UINT32, get(DATA+6,UINT32) - 1852000) # 1000 nm off
    send()
}
```

Example 10.

```
match(CAN1, 0x1F51300, 0x1FFFF00)
{
    if (get(DATA,UINT8) & 0x1F == 1) # 2-nd message in a sequence?
    {
        set(DATA+1, UINT32, get(DATA+1,UINT32) - 1852000) # 1000 nm off
    }
    send()
}
```

Net result of both programs will be the same, but in Example 10 the `match()` subprogram will be executed 3 times for each PGN 0x1F513, as this PGN is sent in a succession of 3 fast-packet frames. The code execution time in the second example is longer, as it has additional operations.

But in the case of Example 9, the Device will receive all three CAN messages first and only then transmit the assembled NMEA message to the program. The time between receiving the first and third messages will be over 1000 microseconds, the period in which two messages on the CAN network must be transmitted. When you call the `send()`, the NMEA 2000 message will be divided into three CAN messages and go to the send queue. The third message will, at the earliest, be sent in 1000 microseconds.

In this way, in Example 10, all messages will be sent approximately 1000 microseconds earlier than in Example 9. In practice, the difference of 1 millisecond may not seem significant, but the longer the NMEA 2000 message, the longer will be the delay. Furthermore, if there is high load on the either of the CAN networks, the program shown in Example 10 has an additional advantage.

X. Debug and logging functionality



Debug mode is not a normal operation mode of the Device. The recording of debug data can lead to additional message delay and loss of some CAN bus messages.

Device supports both raw CAN bus data and program runtime debug information recordings.

To start recording add the following setting to YDNB.CFG file:

```
DIAGNOSTICS=x
```

Where x is log recording duration in seconds.

Insert the card into Device, confirm that the YDNB.CFG file was accepted (Device responds with 3 short GREEN LED flashes) and that the recording had started (3-second long GREEN flash). Do not extract the card during recording as it will damage the card file system. After specified amount of time, log recording will stop and Device will produce a 3-second long RED flash, indicating that all card write operations were finished and card can be safely extracted.

Log file format can be selected by setting:

```
LOG_FORMAT=TEXT or LOG_FORMAT=BINARY
```

Default value is TEXT. If LOG_FORMAT setting is omitted or set to TEXT, Device will record YDNBLOG.TXT file. If YDNBLOG.TXT file already exists on a card, it will be overwritten.

TEXT file will capture all CAN frames for which any `match()` filter yields a match and all CAN frames sent via `send()`. You can also record variable or expression value and type during runtime using a `log()` function:

```
log(expression)
```

Example of the YDNBLOG.TXT file contents:

```
00:30.310 RX CAN1 FILTER 01, 0x09FD0205 1A1A018544FAFFFF
00:30.310 LG CAN1 FILTER 01, INT32 19400
00:30.310 TX CAN2 FILTER 01, 0x09FD0205 1A1A018544FAFFFF
00:30.319 !! FUNC Execution timeout (3000 ms)
```

"RX" and "TX" records designate received and sent CAN frames followed by indication of interface, number of the filter which had a match and CAN frame ID and payload. "LG" record is created by the `log()` function. "!!" record is created on runtime errors.

You can record each message received by the Device into a TEXT log using the following "catch-all" filter:

```
match(ANY, 0, 0)
{
    send()
}
```

Do not forget to delete or comment out all `log()` function calls after debugging session is finished. Even if DIAGNOSTICS is not enabled, `log()` will still be called, resulting in a waste of CPU resources.

If `LOG_FORMAT` is set to `BINARY`, `.CAN` files will be recorded instead. Those files will contain CAN bus data only. All CAN frames received on `CAN1` and `CAN2` interfaces will be recorded, regardless of the filters matches. In contrast to a TEXT log, you cannot use `log()` function to save a value to a BINARY log file.

BINARY logs may be opened, viewed, converted or exported to other formats with our free CAN Log Viewer software, which works on Microsoft Windows, Linux and macOS. You can download it from our web site. The `.CAN` file format is open and is well described in the CAN Log Viewer documentation.

Program code is compiled into a byte code and then executed in a Device "virtual machine". Usually you do not need to worry about program to "assembler code" translation, but if you like to know what is under the hood just add setting:

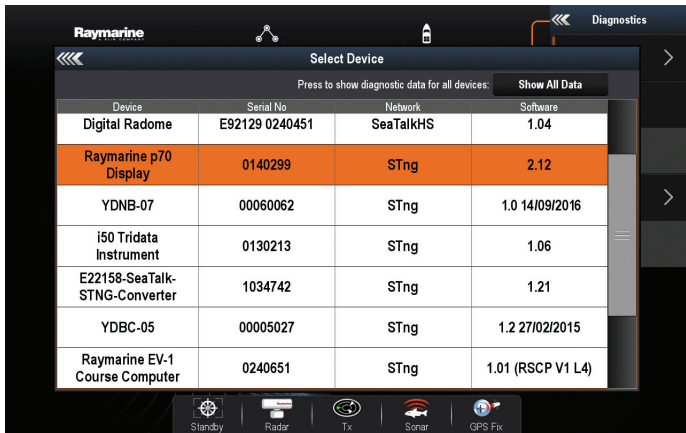
```
DECOMPILER=1
```

to your program, upload it to the Device and you will get `YDNBSAVE.CFG` where every line of your code is accompanied by disassembled byte code.

XI. Firmware Updates

To update the Device firmware, download .ZIP archive from our site and extract BUPDATE.BIN file, take a microSD card formatted with FAT or FAT32 file system, copy BUPDATE.BIN file to the card root folder, power down NMEA 2000, insert card into Device, power up NMEA 2000.

From 5-15 seconds after powering on, the LED will flash 5 times with green light. This indicates that the firmware update is successfully completed.



Device	Serial No	Network	Software
Digital Radome	E92129 0240451	SeaTalkHS	1.04
Raymarine p70 Display	0140299	STng	2.12
YDNB-07	00060062	STng	1.0 14/09/2016
i50 Tridata Instrument	0130213	STng	1.06
E22158-SeaTalk-STNG-Converter	1034742	STng	1.21
YDBC-05	00005027	STng	1.2 27/02/2015
Raymarine EV-1 Course Computer	0240651	STng	1.01 (RSCP V1 L4)

Figure 4. Raymarine c125 MFD devices list with Bridge (YDNB-07)

If the Device already is using the given version of the firmware, or if the Device cannot open the file or the file is corrupted, the boot loader immediately transfers control to the main program. This is done without visual cues.

The Device information including the firmware version is displayed in the list of NMEA 2000 devices (SeaTalk NG, SimNet, Furuno CAN) or in the common list of external devices on the chart plotter (see third line at Figure 4). Usually, access to this list is in the "Diagnostics", "External Interfaces" or "External Devices" menu of the chart plotter

XII. Program protection and encryption

In case you develop programs for a third-party equipment integration, such equipment may have proprietary protocols and you may be under nondisclosure obligations, preventing you to reveal the program code to end-users.

It is also possible to distribute the Device with a program that an end-user cannot modify but can only update with an encrypted file, or even totally lock the Device and exclude any possibilities to read, erase or update the program by the end-user without the master password set by the solution supplier.

Therefore, we have implemented several program protection mechanisms.

Please, refer Application Note YDNB-07-AN-001 available on our web site at:

<http://www.yachtd.com/downloads/>

Appendix A. Troubleshooting

Situation	Possible cause and correction
No LED indication on the Device	<ol style="list-style-type: none"><li data-bbox="306 149 1108 208">1. CAN1 interface is not powered. Device CPU is powered from CAN1 interface. Make sure that power source rails are connected to CAN1 interface.<li data-bbox="306 211 1108 311">2. No power supply on the bus. Check if the bus power is supplied (NMEA 2000 network requires a separate power connection and cannot be powered by a plotter or another device connected to the network).<li data-bbox="306 314 1108 439">3. Loose connection in the power supply circuit. Treat the Device connector with a spray for cleaning electrical contacts. Plug the Device into another connector.
One of the 2-flash LED indication sequence flash is always RED	<ol style="list-style-type: none"><li data-bbox="306 442 1108 522">1. No power supply on the bus. Check if the bus power is supplied (NMEA 2000 network requires a separate power connection and cannot be powered by a plotter or another device connected to the network).<li data-bbox="306 525 1108 594">2. Loose connection in the data circuit. Treat the Device connector with a spray for cleaning electrical contacts. Plug the Device into another connector.<li data-bbox="306 597 1108 698">3. CAN bus or NMEA 2000 network issue. Make sure that both terminators are installed in each network (see Section II). Plug another device into the selected connector and make sure it functions.<li data-bbox="306 701 1108 781">4. Most messages are discarded by hardware filters. Turn any device in the network off and on again (see III.1).
A loaded program is not working properly.	Debug the program. See section X.

Situation	Possible cause and correction
Device is not visible in the NMEA 2000 device list on a NMEA 2000 display device (chartplotter/MFD) while corresponding CAN interface state LED is GREEN	There are problems in the NMEA 2000 network. The network segment is not connected to the plotter or there are missing terminators in the network or the network is too long. Plug another device into the selected connector and make sure it appears in the list of devices on the plotter.

Appendix B. List of NMEA 2000 Messages of the Device

PGNs from the table below are received and sent by the Device independently from any settings, hardware filters (see VII.3) or the program. All received messages, including the messages that are addressed to the Device, are transmitted to the program and may be forwarded to another CAN interface by it. The Device processes messages that are addressed to it after they have been processed by the program (but independently of the program's results). Device's replies to such messages are not passed to the program but are sent directly to both CAN interfaces. See also Section VII.4.

Table 1. Supported NMEA 2000 messages

PGN	Transmit	Receive	Description
59392	Yes	Yes	ISO Acknowledgment
59904	—	Yes	ISO Request
60928	Yes	Yes	ISO Address Claim
126464	Yes	—	PGNs Group List
126996	Yes	—	Product Information

Appendix C. Device Connectors

V+, V- - Battery 12V; CAN H, CAN L - NMEA 2000 data;
SCREEN - Not connected in the Device.

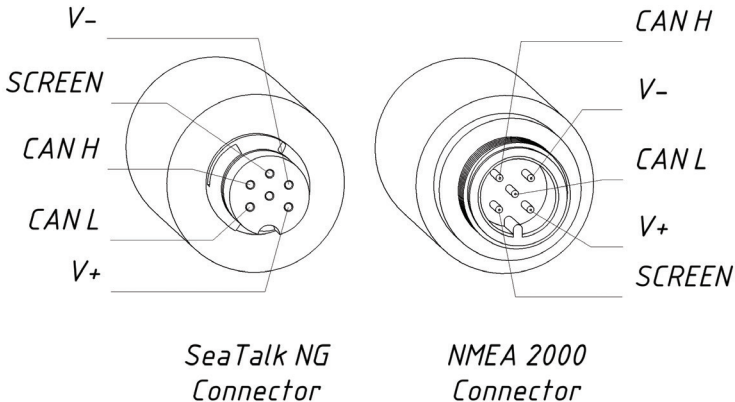


Figure 1. NMEA 2000 connectors of the YDNB-07R (left) and YDNB-07N (right) models

