ssNMEA2000-Multi User's Manual

Version 1.3
Revised April 23rd, 2013
Created by the NMEA 2000 Experts!

ssNMEA2000 Protocol Stack License

READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE OPENING THE PACKAGE CONTAINING THE PROGRAM DISTRIBUTION MEDIA (DISKETTES, CD, ELECTRONIC MAIL), THE COMPUTER SOFTWARE THEREIN, AND THE ACCOMPANYING USER DOCUMENTATION. THIS SOURCE CODE IS COPYRIGHTED AND LICENSED (NOT SOLD). BY OPENING THE PACKAGE CONTAINING THE SOURCE CODE, YOU ARE ACCEPTING AND AGREEING TO THE TERMS OF THIS LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THE TERMS OF THIS LICENSE AGREEMENT, YOU SHOULD PROMPTLY RETURN THE PACKAGE IN UNOPENED FORM, AND YOU WILL RECEIVE A REFUND OF YOUR MONEY. THIS LICENSE AGREEMENT REPRESENTS THE ENTIRE AGREEMENT CONCERNING THE NMEA 2000 PROTOCOL STACK BETWEEN YOU AND SIMMA SOFTWARE, INC. (REFERRED TO AS "LICENSOR"), AND IT SUPERSEDES ANY PRIOR PROPOSAL, REPRESENTATION, OR UNDERSTANDING BETWEEN THE PARTIES.

1. Corporate License Grant. Simma Software hereby grants to the purchaser (herein referred to as the “Client”), a royalty free, non-exclusive license to use the NMEA 2000 protocol stack source code (collectively referred to as the "Software”) as part of Client’s product. Except as provided above, Client agrees to not assign, sublicense, transfer, pledge, lease, rent, or share the Software Code under this License Agreement.

2. Simma Software's Rights. Client acknowledges and agrees that the Software and the documentation are proprietary products of Simma Software and are protected under U.S. copyright law. Client further acknowledges and agrees that all right, title, and interest in and to the Software, including associated intellectual property rights, are and shall remain with Simma Software. This License Agreement does not convey to Client an interest in or to the Software, but only a limited right of use revocable in accordance with the terms of this License Agreement.

3. License Fees. The Client in consideration of the licenses granted under this License Agreement will pay a one-time license fee.

4. Term. This License Agreement shall continue until terminated by either party. Client may terminate this License Agreement at any time. Simma Software may terminate this License Agreement only in the event of a material breach by Client of any term hereof, provided that such shall take effect 60 days after receipt of a written notice from Simma Software of such termination and further provided that such written notice allows 60 days for Client to cure such breach and thereby avoid termination. Upon termination of this License Agreement, all rights granted to Client will terminate and revert to Simma Software. Promptly upon termination of this Agreement for any reason or upon discontinuance or abandonment of Client’s possession or use of the Software, Client must return or destroy, as requested by Simma Software, all copies of the Software in Client’s possession, and all other materials pertaining to the Software (including all copies thereof). Client agrees to certify compliance with such restriction upon Simma Software’s request.

5. Limited Warranty. Simma Software warrants, for Client’s benefit alone, for a period of one year (called the “Warranty Period”) from the date of delivery of the software, that during this period the Software shall operate substantially in accordance with the functionality described in the User's Manual. If during the Warranty Period, a defect in the Software appears, Simma Software will make all reasonable efforts to cure the defect, at no cost to the Client. Client agrees that the foregoing constitutes Client’s sole and exclusive remedy for breach by Simma Software of any warranties made under this Agreement. Simma Software is not responsible for obsolescence of the Software that may result from changes in Client’s requirements. The foregoing warranty shall apply only to the most current version of the Software issued from time to time by Simma Software. Simma Software assumes no responsibility for the use of superseded, outdated, or uncorrected versions of the licensed software. EXCEPT FOR THE WARRANTIES SET FORTH ABOVE, THE SOFTWARE, AND THE SOFTWARE CONTAINED THEREIN, ARE LICENSED "AS IS," AND SIMMA SOFTWARE DISCLAIMS ANY AND ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

6. Limitation of Liability. Simma Software's cumulative liability to Client or any other party for any loss or damages resulting from any claims, demands, or actions arising out of or relating to this License Agreement shall not exceed the license fee paid to Simma Software for the use of the Software. In no event shall Simma Software be liable for any indirect, incidental, consequential, special, or exemplary damages or lost profits, even if Simma Software has been advised of the possibility of such damages. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO CLIENT.

7. Governing Law. This License Agreement shall be construed and governed in accordance with the laws of the State of Indiana.

8. Severability. Should any court of competent jurisdiction declare any term of this License Agreement void or unenforceable, such declaration shall have no effect on the remaining terms hereof.

9. No Waiver. The failure of either party to enforce any rights granted hereunder or to take action against the other party in the event of any breach hereunder shall not be deemed a waiver by that party as to subsequent enforcement of rights or subsequent actions in the event of future breaches.

TABLE OF CONTENTS

Chapter 1

Introduction

ssNMEA2000-Multi is high performance NMEA 2000 protocol stack written in ANSI C. ssNMEA2000-Multi (referred to as ssNMEA2000 from here on out) adheres to both the NMEA 2000 specification and to the software development best practices described in the MISRA C guidelines.

ssNMEA2000 is a modularized design with an emphasis on software readability and performance. ssNMEA2000 is easy to understand and platform independent allowing it to be used on any CPU or DSP with or without an RTOS.

ssNMEA2000 implements the data link layer described in ISO 11783-3, the network management layer described in ISO 11783-5, and the required features (e.g. fast packet support) specified in the NMEA 2000 Main specification. The application layer, described in Appendix B of the NMEA 2000 Main specification, is the responsibility of the end user to implement. Examples of application layer processing are provided in n2000app.c.

Filenames	File Description
n2000.c	Core source file for ssNMEA2000. Do not modify.
n2000.h	Core header file for ssNMEA2000. Do not modify.
n2000tp.c	Transport protocol source file. Do not modify.
n2000tp.h	Transport protocol header file. Do not modify.
n2000fp.c	Fast packet protocol source file. Do not modify.
n2000fp.h	Fast packet protocol header file. Do not modify.
n2000app.c	Application source file for ssNMEA2000. Modification allowed.
n2000app.h	Application header file for ssNMEA2000. Modification allowed.
n2000cfg.h	ssNMEA2000 configuration file. Modification allowed.

Table 1-1: ssNMEA2000 files

Chapter 2

Integration of ssNMEA2000

This chapter describes how to integrate ssNMEA2000 into your application. After this is complete, you will be able to receive and transmit NMEA 2000 messages over CAN. For implementation details, please see the chapters covering the APIs for ssNMEA2000 and ssCAN.

Integration Steps:

1. Develop or purchase a CAN device driver that adheres to the CAN API specified in Chapter 3.

2. Before using any of the NMEA 2000 module features, make sure the CAN driver has been initialized by calling can_init(). Typically it is called shortly after power-on reset and before the application is started.

3. Before using any of the ssNMEA2000 module features, make sure the ssNMEA2000 as been initialized by calling n2000_init(). Typically it is called after can_init() and before the application is started.

4. Call n2000_update at a fixed periodic interval (e.g. every 10 ms). This provides the time base for the NMEA 2000 module. It is recommended that this function be called at least every 25 ms.

5. Set N2000CFG_TICK_PERIOD, in n2000cfg.h, to your systems fixed periodic interval described above in step #4.

6. Set N2000CFG_PORTS_NUM, in n2000cfg.h, to the number of CAN ports in use.

7. Set your “NAME” and “Product Information” fields with n2000_name_set() and n2000_pinfo_set() set. See n2000app_init() for an example.

8. As needed adjust the number and size of the transport protocol buffers.

9. As needed place software in the function n2000app_process(), which is located in n2000app.c, to receive and process NMEA 2000 messages.

10. As needed, call n2000_tx_sf(), n2000_tx_tp(), and n2000_tx_fp() to transmit NMEA 2000 messages.

Chapter 3

ssCAN Application Program Interface

The hardware abstraction layer (HAL) is a software module that provides functions for receiving and transmitting controller area network (CAN) data frames. Because CAN peripherals typically differ from one microcontroller to another, this module is responsible for encompassing all platform depended aspects of CAN communications.

The HAL contains three functions that are responsible for initializing the CAN hardware and handling buffered reception and transmission of CAN frames across multiple ports.

Function Prototype	Function Description
void can_init (void)	Initializes CAN hardware
uint8_t can_rx (uint8_t p, can_t *frame)	Receives CAN frame (buffered I/O)
uint8_t can_tx (uint8_t p, can_t *frame)	Transmits CAN frame (buffered I/O)

Table 3-1: HAL functions

3.1 Data Type Definitions

Data type:

can_t

Description:

can_t is a data type used to store CAN frames. It contains the CAN frame identifier, the CAN frame data, and the size of data. NOTE: If the most significant bit of id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Definition:

```
typedef struct {

  uint32_t id;
  uint8_t buf[8];
  uint8_t buf_len;

} can_t;
```

3.2 Function APIs

can_init

Function Prototype:
void can_init(void);

Description:
can_init initializes the CAN peripheral for reception and transmission of CAN frames at a network speed of 250 kbps. Any external hardware that needs to be initialized can be done inside of can_init. The sample point should be as close to 0.875 as possible, but should not exceed it.

Parameters:
void

Return Value:
void

can_rx

Function Prototype:

uint8_t can_rx (uint8_t p, can_t *frame);

Description:

can_rx checks to see if there is a CAN data frame available in the receive buffer. If one is available, it is copied into the can_t structure that is pointed to by frame. If the most significant bit of frame->id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Parameters:

p: Indicates which port to access.
frame: Points to memory where the received CAN frame should be stored.

Return Value:

1: No CAN frame was read from the receive buffer.
0: A CAN frame was successfully read from the receive buffer.

can_tx

Function Prototype:

uint8_t can_tx (uint8_t p, can_t *frame);

Description:

If memory is available inside the transmit buffer, can_tx copies the memory pointed to by frame to the transmit buffer. If transmission of CAN frames is not currently in progress, then it will be initiated. If the most significant bit of frame->id (i.e. bit 31) is set, it indicates an extended CAN frame, else it indicates a standard CAN frame.

Parameters:

p: Indicates which port to access.
frame: Points to the CAN frame that should be copied to the transmit buffer.

Return Value:

1: No CAN frame was written to the transmit buffer.
0: The CAN frame was successfully written to the transmit buffer.

Chapter 4

ssNMEA2000 Application Program Interface

This chapter describes the application program interface (API) for the NMEA 2000 module.

Function Prototypes	Function Descriptions
void n2000_init (void)	Initializes protocol stack
void n2000_update (void)	Provides periodic time base
void n2000app_process (n2000_t *msg)	Processes received messages
uint8_t n2000_tx_sf (n2000_t *msg)	Transmits a NMEA 2000 single frame message
uint8_t n2000_tx_tp (n2000_t *msg, uint8_t *status)	Transmits a NMEA 2000 transport protocol message
uint8_t n2000_tx_fp (n2000_t *msg, uint8_t seq)	Transmits a NMEA 2000 fast packet message
void n2000_bip_tx_rate_allowed_set (uint8_t p, uint8_t rate)	Sets max allowed transmit rate
uint8_t n2000_bip_tx_rate_max_get (uint8_t p);	Retrieves peak bus load usage.
uint8_t n2000app_sa_get (uint8_t p);	Retrieves next source address
void n2000_name_set (uint8_t p, n2000_name_t *n);	Set internal NAME field.
void n2000_pinfo_set (uint8_t p, n2000_pfino_t *pi);	Set internal Product Info field.

Table 4-1: API functions

4.1 Data Type Definitions

Data type:
n2000_t

Description:
n2000_t is a data type used to store NMEA 2000 messages. It contains the NMEA 2000 message source, destination, PGN, priority, data, and the size of data.

Definition:
```
typedef struct {

  uint32_t pgn;       /* Parameter Group Number. */
  uint8_t *buf;       /* Pointer to data. */
  uint16_t buf_len;   /* Size of data. */
  uint8_t dst;        /* Destination of message. */
  uint8_t src;        /* Source of message. */
  uint8_t pri;        /* Priority of message. */
  uint8_t port;       /* CAN port of message */

} n2000_t;
```

Data type:
n2000_name_t

Description:
n2000_name_t is a data type used to store a CA's NMEA 2000 NAME external to the protocol stack. The function n2000_name_set() must be used to load the NAME field into the stack's internal memory.

Definition:
```
typedef struct {

  uint8_t aac;              /* 1-bit Arbitrary Address Capable */
  uint8_t ind_grp;          /* 3-bit Industry Group */
  uint8_t dev_class_inst;   /* 4-bit Device Class Instance */
  uint8_t dev_class;        /* 7-bit Device Class */
  uint8_t func;             /* 8-bit Function */
  uint8_t func_inst;        /* 5-bit Function Instance */
  uint8_t ecu_inst;         /* 3-bit ECU Instance */
  uint16_t mfg_code;        /* 11-bit Manufacturer Code */
  uint32_t identy_num;      /* 21-bit Identity Number */

} n2000_name_t;
```

Data type:

n2000_pinfo_t

Description:

n2000_pinfo_t is a data type used to store the product information message. The function n2000_pinfo_set() must be used to load the product information field into the stack's internal memory.

Definition:

```
typedef struct {

  uint16_t db_ver;          /* NMEA 2000 Database Version (1.00) */
  uint16_t prd_code;        /* Product Code */
  char *model_id;           /* Model ID */
  char *sw_ver;             /* Software Version */
  char *model_ver;          /* Model Version */
  char *model_scode;        /* Model Serial Code */
  uint8_t cert_level;       /* Certification Level B */
  uint8_t load_eqvalncy; /* Load Equivalency (2 <= 100 milliamps) */

} n2000_pinfo_t;
```

n2000_init

Function Prototype:
void n2000_init (void);

Description:
Initializes the NMEA 2000 module.

Parameters:
void

Return Value:
void

n2000_update

Function Prototype:
void n2000_update (void);

Description:
Provides the periodic time base for the NMEA 2000 module.

Parameters:
void

Return Value:
void

n2000app_process

Function Prototype:

void n2000app_process (n2000_t *msg);

Description:

Processes received NMEA 2000 message. This function is called by the NMEA 2000 module with a complete NMEA 2000 message and is the intended location for the application layer to handle received NMEA 2000 messages. For multipacket messages, this function isn't called until all packets have been received and assembled into a complete NMEA 2000 message.

Parameters:

msg: Pointer to received NMEA 2000 message.

Return Value

void

n2000_tx_sf

Function Prototype:

uint8_t n2000_tx_sf (n2000_t *msg);

Description:

Buffers a NMEA 2000 single frame message for transmission. For messages that are larger than 8 bytes, use n2000_tx_tp() or n2000_tx_fp().

Parameters:

msg: Points to the NMEA 2000 message that should be transmitted.

Return Value:

1: Message was not buffered for transmission.
0: Message was buffered for transmission.

n2000_tx_tp

Function Prototype:

uint8_t n2000_tx_tp (n2000_t *msg, uint8_t *status);

Description:

Buffers a NMEA 2000 transport protocol (i.e. BAM, RTS/CTS) message for transmission. This function is used for non-single-frame messages as defined by the NMEA 2000 application layer.

*status will be equal to N2000TP_INPROCESS while the message is being transmitted, N2000TP_DONE if the message transmission is complete, or N2000TP_FAILED if there was an error.

Parameters:

msg: Points to the NMEA 2000 message that should be transmitted.
status: Points to application RAM.

Return Value:

1: Message was not buffered for transmission.
0: Message was buffered for transmission.

n2000_tx_fp

Function Prototype:

uint8_t n2000_tx_fp (n2000_t *msg, uint8_t seq);

Description:

Buffers a NMEA 2000 fast packet message for transmission. This function is used for non-single-frame messages as defined by the NMEA 2000 application layer.

Parameters:

msg: Points to the NMEA 2000 message that should be transmitted.
seq: 8-bit sequence number ranging from 0 to 255. seq needs to be unique for each fast packet PGN and is to be incremented for each successful transmission of a fast packet message.

Return Value:

1: Message was not buffered for transmission.
0: Message was buffered for transmission.

n2000_bip_tx_rate_allowed_set

Function Prototype:

void n2000_bip_tx_rate_allowed_set (uint8_t p, uint8_t rate);

Description:

In order to implement babbling idiot protection, the NMEA 2000 module tracks how many messages are transmitted by the application in a 250 ms window. If the NMEA 2000 module detects the application has transmitted more messages than is allowed, it will permanently disable transmission. The application should use this function to set the NMEA 2000 module's allowable transmission rate (default of 25%). Set rate to 100% to disable babbling idiot protection.

Parameters:

p: Indicates which port to access.
rate: Max allowed transmission rate. Range from 0 to 100 percent.

Return Value

void

n2000_bip_tx_rate_max_get

Function Prototype:

uint8_t n2000_bip_tx_rate_max_get (uint8_t p);

Description:

In order to implement babbling idiot protection, the NMEA 2000 module tracks how many messages are transmitted by the application in a 250 ms window. The application can use this function to retrieve the max bus load that has been imposed on the bus by the application.

Parameters:

p: Indicates which port to access.

Return Value

0 to 100 percent

n2000app_sa_get

Function Prototype:

uint8_t n2000app_sa_get (uint8_t p);

Description:

This function returns the next source address that the protocol stack should attempt to claim. It is called by the protocol stack and is the responsibility of the application layer to maintain.

The protocol stack calls this function during the initialization process and also when the protocol stack has failed to claim an address. The protocol stack fails to claim an address when it identifies a higher priority ECU that is using the current attempted address.

Since the protocol stack has no knowledge of what additional addresses it may claim, it calls this routine to determine what address is should attempt. When the application layer has no more addresses to claim, this function should return 255.

The source address is a 8-bit field and identifies a unique NMEA 2000 device on the network. Possible values are listed in the NMEA 2000 parent document.

Parameters:

p: Indicates which CAN port the source address should be returned for.

Return Value

Next source address. 255 indicates no more addresses are available.

n2000_name_set

Function Prototype:

void n2000_name_set (uint8_t *p, n2000_name_t *n);

Description:

This function uses the externally pointed to NAME field of n and updates the internal memory of the protocol stack with a correctly formatted 64-bit ISO 11783-5 NAME. Every NMEA 2000/CAN port needs to be assigned a NAME during the initialization processs. These NAMEs may or may not be different depending on the controller application.

The NMEA 2000 name field is 64 bits long and is intended to uniquely describe all ECUs on a NMEA 2000 network (i.e. no two ECUs on a NMEA 2000 network may have the same name field). For more information, see ISO 11783-5.

Parameters:

p: Indicates which CAN port the NAME field is assigned.
n: Points to external version of a NAME field.

Return Value

void

Example

```
n2000_name_t name;

/* table B1 of n2000 */
name.aac = 1;                /* 1-bit Arbitrary Address Capable */
name.ind_grp = 1;            /* 3-bit Industry Group (1 = on-highway) */
name.veh_sys_inst = 0;       /* 4-bit Vehicle System Instance  */
name.veh_sys = 1;            /* 7-bit Vehicle System (1 = tractor) */
name.func = 130;             /* 8-bit Function (130 = data logger)*/
name.func_inst = 0;          /* 5-bit Function Instance */
name.ecu_inst = 0;           /* 3-bit ECU Instance */
name.mfg_code = 402;         /* 11-bit Manufacturer Code (402 = SIMMA) */
name.identy_num = 1009       /* 21-bit Identity Number (see below) */

/* example to set NAME for network 0 */
n2000_name_set( 0, &name );
```

n2000_pinfo_set

Function Prototype:

void n2000_pinfo_set (uint8_t *p, n2000_pinfo_t *pi);

Description:

This function sets the protocol stack's internal product information field to the value pointed to by pi. This function should be called inside of the n2000app_init() function for every NMEA 2000 port.

The protocol stack automatically transmits the product information message every time a request for the production information message is received.

Parameters:

p: Indicates which CAN port the NAME field is assigned.
pi: Points to a product information structure.

Return Value

Void

Chapter 5
Configuration

This chapter describes all configurable items of the NMEA 2000 module. All of these configurations are defined in n2000cfg.h.

NMEA 2000 Name Field
See n2000_name_set() function.

NMEA 2000 Tick Period
The protocol stack needs to be called at a fixed periodic rate. This defines approximately how often the stack will be called. It is defined in units of 0.1 milliseconds. By default, the stack is configured for an update rate of 10 milliseconds.

#define N2000CFG_TICK_PERIOD 100

CAN Ports Count
The protocol stack can support multiple NMEA 2000 networks simultaneously. This defines how many ports the stack will support.

#define N2000CFG_PORTS_NUM 3

Transport Protocol RX Buffer Count
Multipacket NMEA 2000 messages are split into multiple CAN frames and buffered inside of the NMEA 2000 module. This configuration defines how many incoming multipacket messages can be received simultaneously.

#define N2000CFG_TP_RX_BUF_NUM 10

Transport Protocol RX Buffer Size
Multipacket NMEA 2000 messages are split into multiple CAN frames and buffered inside of the NMEA 2000 module. This configuration defines in bytes the largest message which can be received. The maximum buffer size is1,785 bytes.

```
#define N2000CFG_TP_RX_BUF_SIZE       128
```

Transport Protocol TX Buffer Count

Multipacket NMEA 2000 messages are split into multiple CAN frames and buffered inside of the NMEA 2000 module. This configuration defines how many outgoing multipacket messages can be transmitted simultaneously.

```
#define N2000CFG_TP_TX_BUF_NUM        3
```

Transport Protocol TX Buffer Size

Multipacket NMEA 2000 messages are split into multiple CAN frames and buffered inside of the NMEA 2000 module. This configuration defines in bytes the largest message which can be transmitted. The maximum buffer size is1,785 bytes.

```
#define N2000CFG_TP_TX_BUF_SIZE       128
```

Fast Packet RX Buffer Size

Fast packet NMEA 2000 messages are split into multiple CAN frames and buffered inside of the NMEA 2000 module. This configuration defines in bytes the largest fast packet message which can be received. The maximum buffer size is 223 bytes.

```
#define N2000CFG_FP_RX_BUF_SIZE       128
```

Chapter 6

Examples

This chapter gives examples of how to receive/decode NMEA 2000 messages and how to transmit a NMEA 2000 message. Per the NMEA 2000 specification all NMEA 2000 messages have an associated PGN followed by data. The below examples show how to filter based on the PGN of interest and decode the data.

6.1 Receive and Decode NMEA 2000 Messages Example:

```
void
n2000app_process ( n2000_t *msg )
{
    switch( msg->pgn ) {
          /*    -PGN-   -CUSTOMER VARIABLE-              -BUFFER CONVERSION-      */
         case 129284: n2000_dist_to_waypoint     = btou32( &msg->buf[1] ); break;
         case 126992: n2000_time                 = btou32( &msg->buf[4] ); break;
         case 128267: n2000_water_depth          = btou32( &msg->buf[1] ); break;
         case 129025: n2000_latitude             = btou32( &msg->buf[0] );
                      n2000_longitude            = btou32( &msg->buf[4] ); break;
    }
}
```

6.2 Transmit NMEA 2000 Message Example - Single Frame:

```
void
n2000_transmit_example ( void )
{
  n2000_t  msg;
  uint8_t buf[8] = {0,1,2,3,4,5,6,7};

  /* load message */
  msg.pgn = 65215;
  msg.buf = buf;
  msg.buf_len = 8;
  msg.dst = 255;
  msg.pri = 7;
  msg.port = 0;

  /* transmit message */
  if( n2000_tx_sf(&msg ) == 0 )
    printf("Message transmitted\n");
  else
    printf("Message not transmitted\n");
}
```

6.3 Transmit NMEA 2000 Message: Fast Packet

```
void
n2000_transmit_example ( void )
{
  n2000_t  msg;
  static uint8_t seq = 0;
  uint8_t buf[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

  /* load message */
  msg.pgn = 65215;
  msg.buf = buf;
  msg.buf_len = 16;
  msg.pri = 7;
  msg.port = 0;

  /* 255 sends a  to global */
  msg.dst = 255;

  /* transmit message */
  if( n2000_tx_fp(&msg, 0) == 0 )
    seq++, printf("Message transmitted\n");
  else
    printf("Message not transmitted\n");
}
```

6.4 Periodically Request NMEA 2000 Parameters Example:

The below example shows how to transmit requests, for PGNs 65253, 65244, and 65257, periodically using the built-in example function n2000app_tx_request(). The requests are sent every 5 seconds with a 1 second spacing. The below routine assumes the protocol stack's update function is called every 1 millisecond.

```
void
n2000app_update ( void )
{
    static uint16_t time = 0;

    /* transmit requests every 5 seconds on network 0 */
    switch( time++ ) {
        case 1000: n2000app_tx_request( 0,65253, 255); break;
        case 2000: n2000app_tx_request( 0, 65244, 255); break;
        case 3000: n2000app_tx_request( 0, 65257, 255); break;
        case 5000: time = 0;
    }
}
```