ssLIN-Slave
User's Manual

Created by the J1939 Experts!
Visit our LIN Software page.
Version 1.0 - December 11, 2015

ssLIN-Slave 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 J1939 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 LIN 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 breach

TABLE OF CONTENTS

Chapter 1: Introduction

Local Interconnect Network (LIN) is a low cost, low speed network intended for small automotive peripherals that makes use of traditional micro-controller based UARTS. The single wire, bi-directional, half-duplex, asynchronous communication is capable of 19200 baud. The baud rate is determined by the master. A typical network is made up of one master and many slaves. The master is the only node that transmits a break signal, a sync byte (0x55), and then the ID. Depending on the predetermined type of message, the master then transmits the remaining message data or waits to receive an incoming message from the slave that has the corresponding ID.

Filenames	File Description
linSlave.c	Core source file for ssLIN-Slave. Do not modify.
linSlave.h	Core header file for ssLIN-Slave. Do not modify.
linCfgSlave.h	ssLIN-Slave configuration file. Modification allowed.
linSlaveApp.c	Source file for user application additions. Modification allowed.
linSlaveApp.h	Header file for core interface to user functions.
linuart.h	Header file for ssLIN-Slave driver prototypes

Table 1: ssLIN-Slave Files

Chapter 2: Integration of ssLIN-Slave

This chapter describes how to integrate ssLIN-Slave into your system. After this is complete, you will be able to respond to publish and subscribe LIN messages. For implementation details, please see the chapters covering the APIs.

Integration Steps:

1. Develop or purchase a LIN UART device driver which adheres to the ssLIN-Slave API specified in chapter Chapter 3:: , page 6.

2. Before using any of the ssLIN-Slave module features, make sure the stack has been initialized by calling linInitSlave (page 14). This, in turn, calls linuartInit which sets up the UART from step 1 above (page 7). Typically it is called shortly after power-on reset and before the application's main executive is started.

3. Call linSlaveUpdate at a fixed periodic interval (e.g. every 5 ms). This provides the time base for the ssLIN-Slave stack. It is recommended that this function be called at least every 5 ms. See the ssLIN-Slave API, section **5.1.2**, page 15.

4. Configure ssLIN-Slave by editing linCfgSlave.h. See the Configuration chapter Chapter 4:, page 11 for explicit details.

5. Modify and add functionality to linSlaveApp.c This file contains functions that are called during LIN message handling. See chapter **Chapter 5::** , page 14.

Chapter 3: ssLIN-Slave Driver API

The LIN UART driver application program interface (API) is a software module that provides functions for receiving and transmitting LIN bytes. Because UART peripherals typically differ from one microcontroller to another, this module is responsible for encompassing all platform dependent aspects of LIN communications.

The LIN UART Driver API contains four functions that are responsible for initializing the LIN UART hardware and handling buffered reception and transmission of LIN message bytes. This module also has two functions for handling data to/from the EEPROM. Since this a processor specific module, these functions are found here. If your project already has EEPROM driver, simply call your driver from these functions.

Function Prototype	Function Description
void linuartInit (void)	Initializes UART hardware
void linuartSend (uint8_t)	Sends a byte to the LIN transceiver
void linuartSendBreak (void)	Sends a break to the LIN transceiver of at least 13 bit periods
void linRXISR (void)	Receive Interrupt Service Routine. Read one byte or break signal from LIN UART.
void linuartSave (uint8_t *, uint8_t, uint8_t)	Save LIN configuration data to EEPROM
void linuartRestore (uint8_t *, uint8_t, uint8_t)	Restore LIN configuration data from EEPROM

Table 2: LIN UART Driver API functions

3.1 Function APIs

3.1.1 linuartInit

Function Prototype:

```
void linuartInit ( void );
```

Description:
linuartInit initializes the UART peripheral for reception and transmission of LIN data at the desired network speed not to exceed 19200 bps. Any external hardware that needs to be initialized can be done inside of linuartInit. The UART should be configured as no parity, 1 stop bit, and 8 bit data. Receive (only) interrupts should be enabled. See *LIN Specification Package, Revision 2.2A, December 31, 2010* for additional bit timing.

Parameters:
N/A

Return Value:
N/A

3.1.2 linuartSend

Function Prototype:

```
void linuartSend ( uint8_t data );
```

Description:
linuartSend puts the value of its one argument into the transmission queue of the LIN UART.

Parameters:
data: an 8 bit value to be sent via the LIN UART.

Return Value:
N/A

3.1.3 linuartSendBreak

Function Prototype:

```
void linuartSendBreak ( void );
```

Description:
linuartSendBreak generates a break signal of at least 13 bit periods on the output of the LIN UART.

Parameters:
N/A

Return Value:
N/A

3.1.4 linRXISR

Function Prototype:

```
void linRXISR ( void );
```

Description:
linRXISR is an interrupt service routine that is called when a character is received by the LIN UART and the receive interrupt is generated. Due to the nature of the design of the half duplex LIN bus, a transmit interrupt is not needed, only receive. This routine must be able to determine if a break has been detected. For each byte received, the function linPost(uint8_t) must be called with the received byte as the argument. If the received byte was a break, then the global variable linBreak should be set true (non-zero) and linPost called with its argument 0 (zero). If a framing error has been detected, set linFrErr true, else reset to zero (false). See API section 5.1.9 , page 18.

Parameters:
N/A

Return Value:
N/A

3.1.5 linuartSave

Function Prototype:

```
void linuartSave ( uint8_t *buff, uint8_t len, uint8_t adr );
```

Description:

linuartSave is a function that stores the data pointed to by *buff* into the EEPROM at the address given by *adr* and length given by *len*. There are no returned status. This function is called from linSlaveUpdate thread when a configuration command is received from the Master.

Parameters:

buff: a pointer to an 8 bit array to be saved in the EEPROM.

len: an 8 bit value of the number of bytes to save.

adr: an 8 bit value of the offset to save the data in EEPROM.

Return Value:

N/A

3.1.6 linuartRestore

Function Prototype:

```
void linuartRestore ( uint8_t *buff, uint8_t len, uint8_t adr );
```

Description:

linuartRestore is a function that reads the configuration data in EEPROM at address *adr* into an array pointed to by *buff* for length given by *len*. There are no returned status. This function is called from linInitSlave to restore the configuration saved previously. If the EEPROM is erased (first value is 0xFF) then the default configuration is used.

Parameters:

buff: a pointer to an 8 bit array to receive data from EEPROM.

len: an 8 bit value of the number of bytes to read.

adr: an 8 bit value of the offset to read the data in EEPROM.

Return Value:

N/A

Chapter 4: Configuration

This chapter describes all the configurable items of the ssLIN-Slave stack. The majority of the ssLIN-Slave configuration is done in the linCfgSlave.h header file. There is not an automated means to configure this Slave Stack, it is done manually by editing the header file. The message schedule is held in a C structure. The developer must define the number of unique unconditional and event frames that are sent each update period on demand. This header file also declares the frame array, as such, do not include this header file in any other modules.

4.1 Frames

All frames, Unconditional and event frames, are defined in linCfgSlave.h. The total number of frames is defined in macro LIN_NumFrames.

```
/* Schedule Table Size, MUST DEFINE */
#define LIN_NumFrames              3  /* number of frames in Main Table */
```

Timing is configured in the linCfgSlave.h. The value of LIN_TicPeriod is the number of microseconds between calls to the state machine, linSlaveUpdate(). The state machine must be called periodically and with time precision between 1 and 5000 microseconds with 1000 the norm. The value of LIN_FramePeriod is the number of milliseconds the state machine allocates for each frame. Generally the frame period is more than 5 milliseconds.

```
/* SET PERIODS here, MUST DEFINE */
#define LIN_TicPeriod              1000          /* microseconds       */
#define LIN_FramePeriod            45            /* milliseconds       */
```

Note: the LIN_FramePeriod allows the state machine to calculate a time out and return to listening.

The allowable LIN frames are defined in a C struct array, auto-initializer declaration in the linCfgSlave.h header file. Some fields are changed during run time, thus the structure must live in RAM and not defined in ROM. A sample is shown here:

```
linSched_t linMainSched[ LIN_NumFrames ] = {

  { /* frame 1 */
    LIN_SUBSCRIBE,          /* publish / subscribe / event                  */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)   */
    FALSE,                  /* no data yet                                  */
    LINprotectID(0x10),     /* ID                                           */
    2, {0,0,0,0,0,0,0,0}    /* data length and area                         */
  },
  { /* frame 2 */
    LIN_PUBLISH,            /* publish / subscribe / event                  */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)   */
    FALSE,                  /* no data yet                                  */
    LINprotectID(0x15),     /* ID                                           */
    2, {0,0,0,0,0,0,0,0}    /* data length and area                         */
  },
  { /* frame 3 */
    LIN_EVENT,              /* publish / subscribe / event                  */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)   */
    FALSE,                  /* no data yet                                  */
    LINprotectID(0x25),     /* ID                                           */
    4, {0,0,0,0,0,0,0,0}    /* data length and area                         */
  }
};
```

The sample main schedule table shown above defines three frames, one of each type.

The first frame (frame 1) is an unconditional, subscribe frame. This will be sent to the Master when a request is received for ID 0x10. The terms Publish and Subscribe are master centric. That is, a publish message is always from the master, a subscribe message is always to the master.

The second frame (frame 2) is an unconditional, publish frame. This will cause the slave to receive the data from the master if ID 0x15 is received.

The third frame (frame 3) is an event frame. Event frames are a type of subscribe frame. This is the response frame for the master request with ID 0x25. Multiple slaves can support this same frame. The slave only responds if the ready flag is TRUE. If more than one respond, then the master will request the unconditional frames from each slave.

Main Schedule Table Fields	
Frame Type	LIN_PUBLISH or LIN_SUBSCRIBE or LIN_EVENT. Publish type frames are received from the master, and subscribe type frames are sent to the master. The Event type allows for multiple devices to share this Subscribe only frame period and the master will mitigate collisions.
Version	LIN_ENHANCED or LIN_CLASSIC. A boolean value to use version 1.3 or earlier CRC calculation method or later enhanced method.
Status	Set to FALSE. A boolean indicator of received Subscribed message data or Event ready.
ID	The PID (protected frame identifier) for the slave. Use the macro LINprotectID(frame_ID) to set the parity bits in the message field.
Data length	1 to 8. Set the length of the data in the message.
Data	Leave as eight zeros. The data buffer for the frame.

Table 3: Main Schedule Fields

4.2 Event Frames

Event frames are subscribe (send to master) frames that happen occasionally from multiple nodes. The LIN master provides a means to read from several nodes during one time slot to provide network efficiency. During the slot for the defined event frame any node on the network cluster that has data for the defined ID can send. If multiple nodes send at the same time, a collision occurs. To mitigate the collision, the master will then send all individual PIDs (protected frame IDs) associated with the event to determine which nodes had collided. The related unconditional subscribe frame must also be defined in the table to meet the LIN specification.

Chapter 5: ssLIN-Slave API

This chapter describes the application program interface (API) for the LIN Master stack. Use the following API functions to manage your messages:

Function Prototype	Function Description
void linInitSlave (void)	Initializes LIN stack and hardware
void linSlaveUpdate (void)	Processes received frame
uint8_t linReadUCPublish (uint8_t fidx, uint8_t *data)	Read incoming data from master output frame
uint8_t linSetUCSubscribe(uint8_t fidx, uint8_t *data)	Write outgoing frame data
uint8_t linSetEvent(uint8_t fidx, uint8_t *data)	Write event frame data
uint8_t linIsBusy(void)	Test state of LIN machine
uint8_t linRespError(void)	Read the communication error state
uint8_t linIsSleepy(void)	Return status of sleep
void linPost(uint8_t data)	Post received character to LIN core

Table 4: API functions

5.1 Function APIs

5.1.1 linInitSlave

Function Prototype:

```
void linInitSlave ( void );
```

Description:

Call this once before anything else to initialize the UART and the LIN slave stack. This function also reads the configuration saved in EEPROM.

Parameters:

N/A

Return Value:

N/A

5.1.2 linSlaveUpdate

Function Prototype:

```
void linSlaveUpdate ( void );
```

Description:

This function must be called periodically and with precision with the period defined in linCfgSlave.h. Once every 5 milliseconds is a good choice. This drives the message processing state machine and message time outs.

Parameters:

N/A

Return Value:

N/A

5.1.3 linReadUCPublish

Function Prototype:

```
uint8_t linReadUCPublish( uint8_t fidx, uint8_t *data );
```

Description:

Read the data that was published from the master that was sent in a periodically scheduled slot defined as an unconditional published frame.

Parameters:

fidx: the index of the frame defined in the main schedule (linMainSched). Indexes are zero based.

data: a byte buffer containing your new data to be read from the frame. This must be at least as long as the data defined in the table.

Return Value:

0: (FALSE) successful

1: found, but data is old
2: error, not found

5.1.4 linSetUCSubscribe

Function Prototype:

```
uint8_t linSetUCSubscribe( uint8_t fidx, uint8_t *data );
```

Description:

Write the data to be sent to the master in this unconditional subscribe frame.

Parameters:

fidx: the index of the frame defined in the main schedule (linMainSched). Indexes are zero based.

data: a byte buffer containing your new data to be saved in the frame. This must be at least as long as the data defined in the table.

Return Value:

0: (FALSE) successful
1: error, not found
2: error, busy sending this frame

5.1.5 linSetEvent

Function Prototype:

```
uint8_t linSetEvent( uint8_t fidx, uint8_t *data );
```

Description:

Write the data to be sent in a periodically scheduled slot defined as an event frame.

Parameters:

fidx: the index of the frame defined in the main schedule (linMainSched). Indexes are zero based.

data: a byte buffer containing your new data to be saved in the frame. This must be at least as long as the data defined in the table.

Return Value:
0: (FALSE) successful
1: error, frame not found
2: error, busy sending this frame

5.1.6 linIsBusy

Function Prototype:

```
uint8_t linIsBusy( void );
```

Description:
Returns the status of transmission of data. This is usually called after a busy error from one of the set functions (linSetUCSubscribe(), linSetEvent()). Since the message state is driven by the receive interrupt, the application can spin on this status until false.

Parameters:
N/A

Return Value:
0: (FALSE) not busy
1: (TRUE) busy

5.1.7 linRespError

Function Prototype:

```
uint8_t linRespError( void );
```

Description:
This function reads and clears the error status of the LIN bus. In this way, a slave can report the status (in a Boolean fashion) of any data errors on the bus. The user application then can return this value in any of its subscribe frames.

Parameters:
N/A

Return Value:
0: (FALSE) no errors
1: (TRUE) an error has occurred

5.1.8 linIsSleepy

Function Prototype:

```
uint8_t linIsSleepy( void );
```

Description:
This function returns the state of the LIN core. A TRUE return means that the LIN slave has received the sleep command, or that the LIN bus has been idle for more than 4 seconds. The application should perform the processor required steps to enter into sleep mode. The processor should also be configured to wake on a change to the LIN bus from idle to active (high to low transition).

Parameters:
N/A

Return Value:
0: (FALSE) not ready for sleep
1: (TRUE) ready for sleep

5.1.9 linPost

Function Prototype:

```
void linPost( uint8_t data );
```

Description:
This function must be called with the data received in the LIN UART receive ISR (interrupt service routine). It is also expecting the global variables linBreak, and linFrErr to be set true (non-zero) or false (zero). The global variable linBreak is set if a break condition is detected. The global variable linFrErr is set if a framing error is detected. Otherwise these variables should be cleared.

Parameters:

data: the byte received by the UART. The value shall be zero for break condition.

Return Value:

N/A

5.2 Diagnostic API

If expanding on the diagnostic API be sure to configure LIN_DIAG_BUFFERSIZE in linCfgSlave.h. Otherwise, set this value to a low value of 8 to conserve memory allocation and remove the sample code from handlers in linSlaveApp.c.

```
#define LIN_DIAG_BUFFERSIZE  24  /* set the maximum size of request msg + 2 */
```

Several diagnostic callback functions are defined in linSlaveApp.c. This module contains all the optional diagnostic functions and is intended for customer modification. These functions are called in response to messages received from the LIN master. The functions are called from linSlaveUpdate and are not during interrupt processing. In addition, the product ID, functional ID, variant ID, and serial number are defined in this module. If some of these IDs are not stored in program space, then code must be added to recall these values and store them in the temporary variables provided. See function linSlaveApp_init, below.

Function Prototype	Function Description
void linSlaveApp_init(void)	Initializes application level variables
uint8_t linSlaveApp_readByIdent(uint8_t id)	Response to diagnostic function B2
void linSlaveApp_dataDump(uint8_t *buff)	Response to diagnostic function B4
uint8_t linSlaveApp_UDSreadByIdent(uint16 id)	Response to diagnostic function 22

Table 5: Diagnostic Functions

5.2.1 linSlaveApp_init

Function Prototype:

```
void linSlaveApp_init( void );
```

Description:

This function retrieves the product ID, function ID, variant ID, serial number, and initial NAD from storage. The supplied code reads these values from constant arrays (program memory). If these values are not to be stored in program memory due to the method of the user's final manufacturing process, then the user must supply code to move the values to the supplied temporary variables. Note, the LIN specification has

fixed sizes for these values as indicated in the source code.

Parameters:
N/A

Return Value:
N/A

5.2.2 linSlaveApp_readByIdent

Function Prototype:

```
uint8_t linSlaveApp_readByIdent( uint8_t id );
```

Description:
This function is called in response to diagnostic code B2. Identification codes zero (0) and one (1) are handled by the LIN core and do not need to be addressed in this function. This function handles any of the user identified codes 32 through 63. If a code is handled, set the return value to zero (0). Add your handler code in the switch statement provided, case 32 is shown as an example.

Response data must be moved to the array *linDiagBuffer*[]. The first byte of the array is the NAD, which is found in the variable *linNAD*. The second byte must be the response ID (RID) which for this function shall be 0xF2. The next bytes are your user data. Finally, place the number of user data bytes plus 1 into the variable *linDiagRqstFlag*. This will signal the core to send your data when the master requests a response. Note, your entire response must be equal or less than the size specified in LIN_DIAG_BUFFERSIZE.

Parameters:
id: identification number requested by the master.

Return Value:
0: successful, identification code handled.
1: identification code not handled, error response shall be sent.

5.2.3 linSlaveApp_dataDump

Function Prototype:

```
void linSlaveApp_dataDump( uint8_t *buff );
```

Description:

This function is called in response to diagnostic code B4. If applicable, the user may fill in the buffer *linDiagBuffer*[] with the user data. The first byte of the array is the NAD, which is found in the variable *linNAD*. The second byte must be the response ID (RID) which for this function shall be 0xF4. The next bytes are your user data. Finally, place the number of user data bytes plus 1 into the variable *linDiagRqstFlag*. This will signal the core to send your data when the master requests a response. Note, your entire response must be equal or less than the size specified in LIN_DIAG_BUFFERSIZE.

This function has example code to read the EEPROM based on the address found in the first data byte of the passed request. The user must remove this response if it is not desired. The passed request data is available through the pointer passed to this function. If no response is desired, leave *linDiagRqstFlag* unchanged.

Parameters:

buff: a pointer to the 8 byte message sent by the master. Format: NAD, PCI, SID, data. PCI is 6, SID is 0xB4, data is up to 5 bytes.

Return Value:

N/A

5.2.4 linSlaveApp_UDSreadByIdent

Function Prototype:

```
uint8_t linSlaveApp_UDSreadByIdent( uint16_t id );
```

Description:

This function is called in response to diagnostic code 22, UDS (ISO14229). This

message defines a 16 bit ID which is passed to this function. Two IDs are predefined in the specification; 0 shall return the hardware version, 1 shall return the software version. The example code provided shows the hardware version handler. It is up to the customer to provide the handler code for these two identifiers. Add your handlers to the switch statement provided.

Response data must be moved to the array *linDiagBuffer*[]. The first byte of the array is the NAD, which is found in the variable *linNAD*. The second byte must be the response ID (RID) which for this function shall be 0x62. The next bytes are your user data. Finally, place the number of user data bytes plus 1 into the variable *linDiagRqstFlag*. This will signal the core to send your data when the master requests a response. Note, your entire response must be equal or less than the size specified in LIN_DIAG_BUFFERSIZE.

Parameters:

id: identification number requested by the master.

Return Value:

0: successful, identification code handled.
1: identification code not handled, error response shall be sent.