ssLIN-Master
User's Manual

Created by the J1939 Experts!
Visit our LIN Software page.
Version 1.1 - December 11, 2015

ssLIN-Master 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
linMaster.c	Core source file for ssLIN-Master. Do not modify.
linMaster.h	Core header file for ssLIN-Master. Do not modify.
linCfg.h	ssLIN-Master configuration file. Modification allowed.
linuart.h	Header file for LIN driver prototypes

Table 1: ssLIN-Master Files

Chapter 2: Integration of ssLIN-Master

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

Integration Steps:

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

2. Before using any of the ssLIN-Master module features, make sure the stack has been initialized by calling linInitMaster (page 15). 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 linMasterUpdate at a fixed periodic interval (e.g. every 1 ms). This provides the time base for the ssLIN-Master stack. It is recommended that this function be called at least every 5 ms. See the ssLIN-Master API, section 5.1.2, page 16.

4. Configure the LIN Master by editing linCfg.h. See chapter Chapter 4:: Configuration, page 10 for explicit details.

5. Use the ssLIN-Master API to send and receive LIN messages. See chapter Chapter 5:, page 15. Also, if diagnostic information needs to be passed, use the Diagnostic API in section 5.2, page 21.

Chapter 3: ssLIN-Master 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.

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.

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 , page 19.

Parameters:
N/A

Return Value:
N/A

Chapter 4: Configuration

This chapter describes all the configurable items of the ssLIN-Master stack. The majority of the ssLIN-Master configuration is done in the linCfg.h header file. There is not an automated means to configure this Master 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.

4.1 Frames

Unconditional and event frames are always sent each update period. See LIN_NumFrames. Also, the number of sporadic frames must be defined, see LIN_NumSporFrames. This figure may be set to zero if no sporadic frames are used. A sporadic frame is a publish type (sent by master) message that is only sent if new data is present in its buffer. More on sporadic frames later.

```
/* Schedule Table Sizes, MUST DEFINE */
#define LIN_NumFrames            3  /* number of frames in Main Table */
#define LIN_NumSporFrames        1  /* set to 0 if no sporadic frames    */
```

Timing is configured in the linCfg.h. The value of LIN_TicPeriod is the number of microseconds between calls to the state machine, linMasterUpdate(). 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 master 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          20          /* milliseconds       */
```

Note: the total update period is the LIN_FramePeriod times the number of defined unconditional and event frames. The update period is increased by one frame period if one or more sporadic frames are defined.

The main schedule is defined in a C struct array, auto-initializer declaration in the linCfg.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_UNCONDITIONAL,      /* frame type                                  */
    LIN_SUBSCRIBE,          /* publish (send) or subscribe (receive)       */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)  */
    0,0,                    /* no event collision table                    */
    FALSE,                  /* no data yet                                 */
    LINprotectID(0x10),     /* ID                                          */
    8, {0,0,0,0,0,0,0,0}    /* data length and area                        */
  },
  { /* frame 2 */
    LIN_UNCONDITIONAL,      /* frame type                                  */
    LIN_PUBLISH,            /* publish (send) or subscribe (receive)       */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)  */
    0,0,                    /* no event collision table                    */
    FALSE,                  /* no data yet                                 */
    LINprotectID(0x15),     /* ID                                          */
    8, {0,0,0,0,0,0,0,0}    /* data length and area                        */
  },

  { /* frame 3 */
    LIN_EVENT,              /* frame type                                  */
    LIN_SUBSCRIBE,          /* publish (send) or subscribe (receive)       */
    LIN_ENHANCED,           /* checksum Enhanced (v2.x) or Classic (v1.x)  */
    2, linEvent25Table,     /* size and event collision table pointer      */
    FALSE,                  /* no data yet                                 */
    LINprotectID(0x25),     /* ID                                          */
    8, {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 cause the master to request data from slave ID 0x10.

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

The third frame (frame 3) is an event frame. Event frames must be type subscribe. This will cause the master to request data from all slaves with ID 0x25. If more than one respond, then a collision mitigation schedule must be defined in fields four and five. More on event frames later.

Main Schedule Table Fields	
Frame Type	LIN_UNCONDITIONAL or LIN_EVENT. Unconditional frame type is the standard LIN frame. The Event type allows for multiple devices to share this Subscribe only frame period and the master will mitigate collisions.
Publish/Subscribe	LIN_PUBLISH or LIN_SUBSCRIBE. A boolean value that directs the master to send (publish) or receive (subscribe)
Version	LIN_ENHANCED or LIN_CLASSIC. A boolean value to use version 1.3 or earlier CRC calculation method or later enhanced method.
Event Schedule Size	Zero for Unconditional frames, 1 or more for Event frames. This value defines the number of frames in the unique collision schedule table for this particular event.
Event Schedule Table	Zero for Unconditional frames, or a pointer to the collision schedule table for this particular event.
Status	Set to FALSE. A boolean indicator of received Subscribed message data.
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 Sporadic Frames

These frames are a type of publish (send only) frames that do not need to be sent every update period. This allows the master to optimize its loop time with other messages. Sporadic frames are sent whenever there is a new data to publish. The frames are configured in the table in priority order with the highest (most important) at the beginning. A time slot is allocated for the sporadic frames if one or more exist. If so, then at that time, the ready status of each sporadic frame is checked in order from the beginning of the table. Only one sporadic frame is published during the designated time slot and its ready flag is reset.

Sporadic Schedule Table Fields	
Ready	Set to FALSE, this flag is set when data is posted to the frame. See linSetSporadic(...)
Version	LIN_ENHANCED or LIN_CLASSIC. A boolean value to use version 1.3 or earlier CRC calculation method or later enhanced method.
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 4: Sporadic Fields

4.3 Event Frames

Event frames are subscribe (read only) 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, the any node on the network 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 identifiers) associated with the event to determine which nodes had collided. To do this, a schedule table associated for each event frame must be defined and contain all PIDs that participate in that event. Here is an example of a schedule table for event frame 0x25:

```
/* table for event frame ID 0x25 */
/* define HERE and reference and size in MAIN table */
linEventCollisionTable_t linEvent25Table[2] = {
  { /* frame 1*/
    LIN_ENHANCED,          /* checksum Enhanced (v2.x) or Classic (v1.x)  */
    FALSE,                 /* no data                                     */
    LINprotectID(0x32),    /* ID                                          */
    8, {0,0,0,0,0,0,0,0}   /* data length and area                        */
  }
  ,{ /* frame 2*/
    LIN_ENHANCED,          /* checksum Enhanced (v2.x) or Classic (v1.x)  */
    FALSE,                 /* no data                                     */
    LINprotectID(0x33),    /* ID                                          */
    8, {0,0,0,0,0,0,0,0}   /* data length and area                        */
  }
};
```

In this example there are two unconditional frames defined for event frame 0x25; frame 0x32 and frame 0x33. These frames return the same data and format as a request for 0x15, but only from the specific slave.

Event Collision Schedule Table Fields	
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.
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 5: Event Collision Fields

Chapter 5: ssLIN-Master 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 linInitMaster (void)	Initializes LIN stack and hardware
void linMasterUpdate (void)	Provides periodic time base
uint8_t linSetUCPublish (uint8_t id, uint8_t *data)	Updates unconditional output frame
uint8_t linSetSporadic (uint8_t id, uint8_t *data)	Updates sporadic output frame
uint8_t linReadUCSubscribe(uint8_t id, uint8_t *data)	Read incoming data from a frame
uint8_t linReadEvent(uint8_t *id, uint8_t *eid, uint8_t *data)	Read event data from a frame
void linSetSchedEnable(void)	Start the LIN frame schedule
void linResetSchedEnable(void)	Suspend the LIN frame schedule
void linPost(uint8_t data)	Called by LIN UART receive ISR

Table 6: API functions

5.1 Function APIs

5.1.1 linInitMaster

Function Prototype:

```
void linInitMaster ( void );
```

Description:
Call this once before anything else to initialize the UART and the LIN master stack.

Parameters:
N/A

Return Value:
N/A

5.1.2 linMasterUpdate

Function Prototype:

```
void linMasterUpdate ( void );
```

Description:

This function must be called periodically and with precision with the period defined in linCfg.h. Once per millisecond is a good choice.

Parameters:

N/A

Return Value:

N/A

5.1.3 linSetUCPublish

Function Prototype:

```
uint8_t linSetUCPublish( uint8_t id, uint8_t *data );
```

Description:

Update the data to be sent in a periodically scheduled slot defined as an unconditional published frame.

Parameters:

id: the non-protected frame ID is the key to identify the frame.

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: (TRUE) not found

5.1.4 linSetSporadic

Function Prototype:

```
uint8_t linSetSporadic( uint8_t id, uint8_t *data );
```

Description:
Update the data to be sent in a periodically scheduled slot defined as a sporadic frame.

Parameters:
id: the non-protected frame ID is the key to identify the frame.

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: (TRUE) not found

5.1.5 linReadUCSubscribe

Function Prototype:

```
uint8_t linReadUCSubscribe( uint8_t id, uint8_t *data );
```

Description:
Read the data received in a periodically scheduled slot defined as an unconditional subscribe frame.

Parameters:
id: the non-protected frame ID is the key to identify the frame.

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: found, but data unchanged since last read
2: not found

5.1.6 linReadEvent

Function Prototype:

```
uint8_t linReadEvent( uint8_t *id, uint8_t *eid, uint8_t *data );
```

Description:
Read the data received in a periodically scheduled slot defined as an event frame. Call this repeatedly to retrieve all received events (until TRUE).

Parameters:
id: a pointer to a variable to hold the event ID of the received frame. If null, argument ignored.

eid: a pointer to a variable to hold the frame ID of the unconditional frame. If null, argument ignored.

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: (TRUE) no events found or received

5.1.7 linSetSchedEnable

Function Prototype:

```
void linSetSchedEnable( void );
```

Description:
Call this to start the main schedule. This is enabled by default and only is needed if linRestSchedEnable() has been called.

Parameters:
N/A

Return Value:
N/A

5.1.8 linResetSchedEnable

Function Prototype:

```
void linResetSchedEnable( void );
```

Description:
Use to suspend the main schedule. Useful for diagnostics. Restart the schedule with linSetSchedEnable.

Parameters:
N/A

Return Value:
N/A

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 using the diagnostic API be sure to configure LIN_DIAG_BUFFERSIZE in linCfg.h. Otherwise, set this value to a low number to conserve memory allocation.

```
#define LIN_DIAG_BUFFERSIZE  100  /* set the maximum size of request msg + 2 */
```

Function Prototype	Function Description
uint8_t linSendMessage(uint8_t *buf, uint16_t cnt)	Schedules a diagnostic message to be sent
uint8_t linReceiveMessage(void)	Schedules a diagnostic response
uint8_t linRxStatus(uint8_t **buff, uint16_t *sz)	Reads the status and data of a response

Table 7: Diagnostic Functions

5.2.1 linSendMessage

Function Prototype:

```
uint8_t linSendMessage( uint8_t *buf, uint16_t cnt );
```

Description:

Schedule a diagnostic master message (ID 0x3C) to be sent. This message uses the TL (transport layer) and can accommodate messages as long as the buffer defined as LIN_DIAG_BUFFERSIZE in linCfg.h.

Parameters:

buf: a byte array that contains the data to be send. It must contain the destination NAD as byte 0 and the SID as byte 1

cnt: the size of the buffer sent which includes the NAD and SID

Return Value:

0: successful
4: buffer size too large
8: master busy with previous diagnostic tx, try again
9: buffer size too small

5.2.2 linReceiveMessage

Function Prototype:

```
uint8_t linReceiveMessage( void );
```

Description:

Schedule a diagnostic master response request (ID 0x3D) to be sent. This message uses the TL (transport layer) and can accommodate messages as long as the buffer defined as LIN_DIAG_BUFFERSIZE in linCfg.h. Generally used if linSendMessage(...) has been called and can be called immediately following linSendMessage(...).

Parameters:

N/A

Return Value:

0: successful
7: master busy with previous diagnostic rx

5.2.3 linRxStatus

Function Prototype:

```
uint8_t linRxStatus( uint8_t **buff, uint16_t *sz );
```

Description:

Read the received response from the master response request. This should be called after the linReceiveMessage() has been called. Some time should be allowed to receive the frame.

Parameters:

buff: a pointer to a pointer variable to hold the address of the global diagnostic buffer. Memory savings is afforded to access the buffer directly. The buffer will contain the NAD at byte 0 and the RSID at byte 1, followed by the data. A null value ignores this argument.

sz: a pointer to a variable to hold the received size of the data in the buffer, including the

NAD and RSID. A null value ignores this argument.

Return Value:

1: busy, try again
2: message complete, message is ready to be read
3: error in message, format / size error
4: error in message, message larger than buffer
5: time out