To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

H8/300H Tiny Series H8/36049 Group

LIN (Local Interconnect Network): Master Volume

Introduction

LIN (Local Interconnect Network): Master Volume provides examples of settings and usage of the on-chip peripheral functions of H8/300H Tiny Series H8/36049 Group microcomputers to implement communications according to the LIN protocol. This note is provided as a reference to help users in software and hardware design.

The operation of programs, circuits and other items in this application note have been confirmed. However, be sure to confirm the operation before actual usage.

Target Device

H8/300H Tiny Series H8/36049F

Contents

1. Overview of LIN Communications Systems

This section gives an overview of LIN communications on systems that incorporate the sample LIN communications software library (hereinafter referred to as the LIN2.0 library or the library) described in this application note.

1.1 Connection to the LIN Bus

A system connected to a network on a LIN bus via a LIN bus interface circuit (or LIN transceiver) is able to transmit header frames as the master node, as well as transmit and receive response frames.

1.1.1 System Configuration

Figure 1 shows an example of how a network system is configured on a LIN bus.

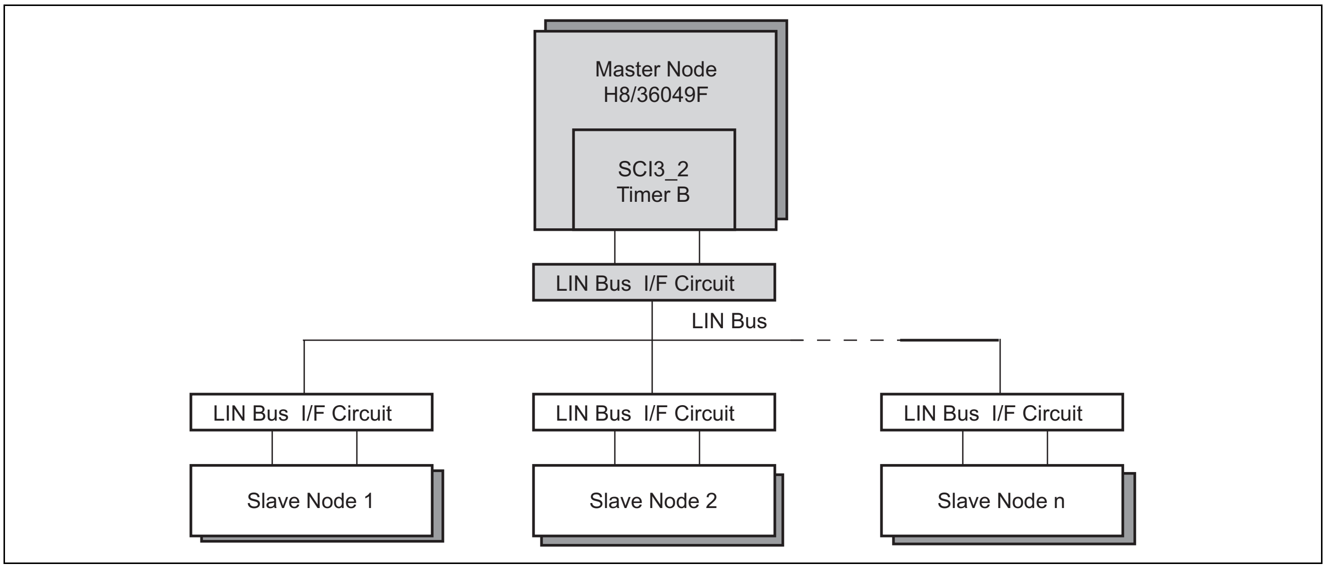

Figure 1 System configuration

1.1.2 Resource Usage

Resources of the H8/36049F for use in this application note are listed in table 1.

Table 1 CPU resources used in master node operation

Function		Pin function (Pin No.)	Usage	Description/Comment
I/O port pin		P60 (42)	LIN transceiver control	LIN transceiver is enabled or disabled by the output of this I/O pin (high and low, respectively). The user must set the pin to be an output at the high level after a reset.)
SCI3 (Channel-2)	Transmission	TXD_2 (72)	Transmission of header and response frames, output of wake-up signal	Asynchronous mode, 8-bit data length, no parity bit, 1-stop bit (with start bit added), LSB first
	Reception	RXD_2 (71)	Reception of response frames	
			Detection of errors in communications	Module’s internal error detection function
Timer B1		-	Measurement of break delimiter and wake-up signal periods	Measurement of break period along with the communication speed

1.2 Overview of LIN Communication

This section gives an overview of the various frames transmitted and received in the LIN communications protocol.

1.2.1 Unconditional Frame

An unconditional frame is always transmitted and received regardless of any updated signal values.

The node that transmits a response to a header can be a master or slave node. Also, the node that receives the response can be a master or slave node.

Sequences for unconditional frames are illustrated in figure 2.

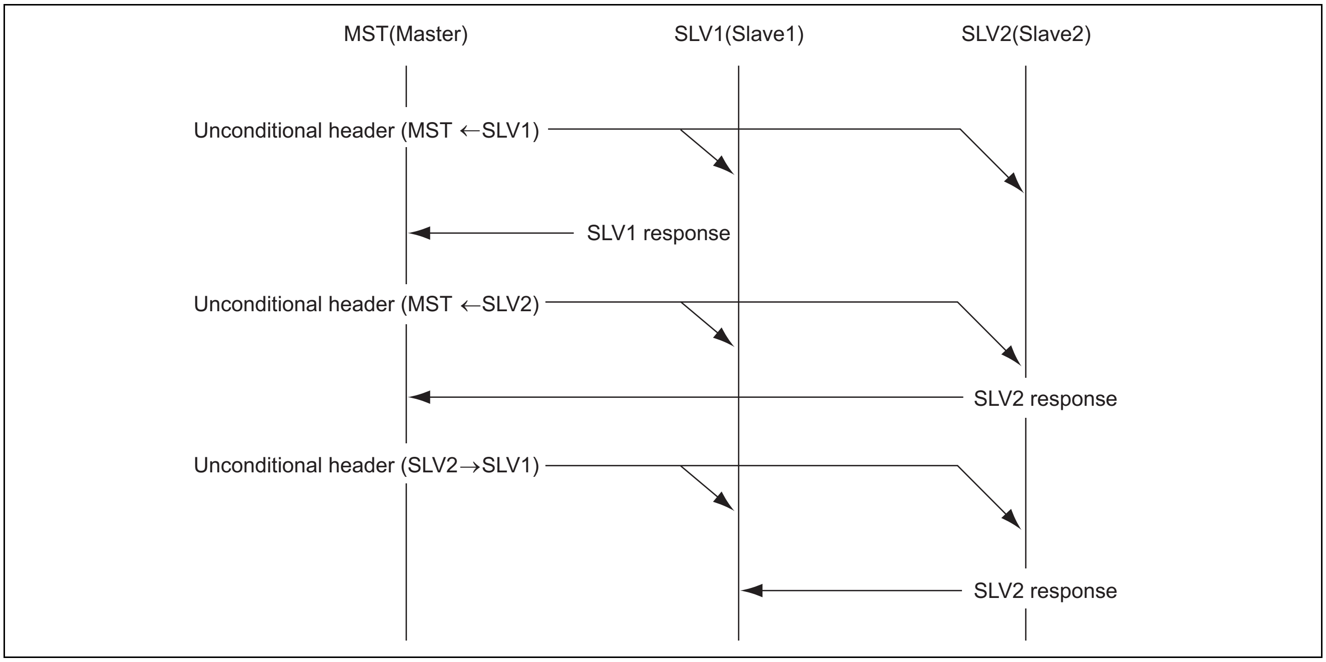

Figure 2 Sequences for unconditional frames

1.2.2 Event-Triggered Frame

An event-triggered frame is transmitted from a master node and received by a slave node in order to confirm the availability of an update to the value of a signal.

Only those slave nodes with updated signal values transmit responses to the header. The transmission of responses by several slave nodes may lead to a collision. When a collision occurs, the master node sends requests for the confirmation of signal values to all of the slave nodes via an unconditional frame. On the other hand, the master node is the only node that receives the responses.

Sequences for event-triggered frames are illustrated in figure 3.

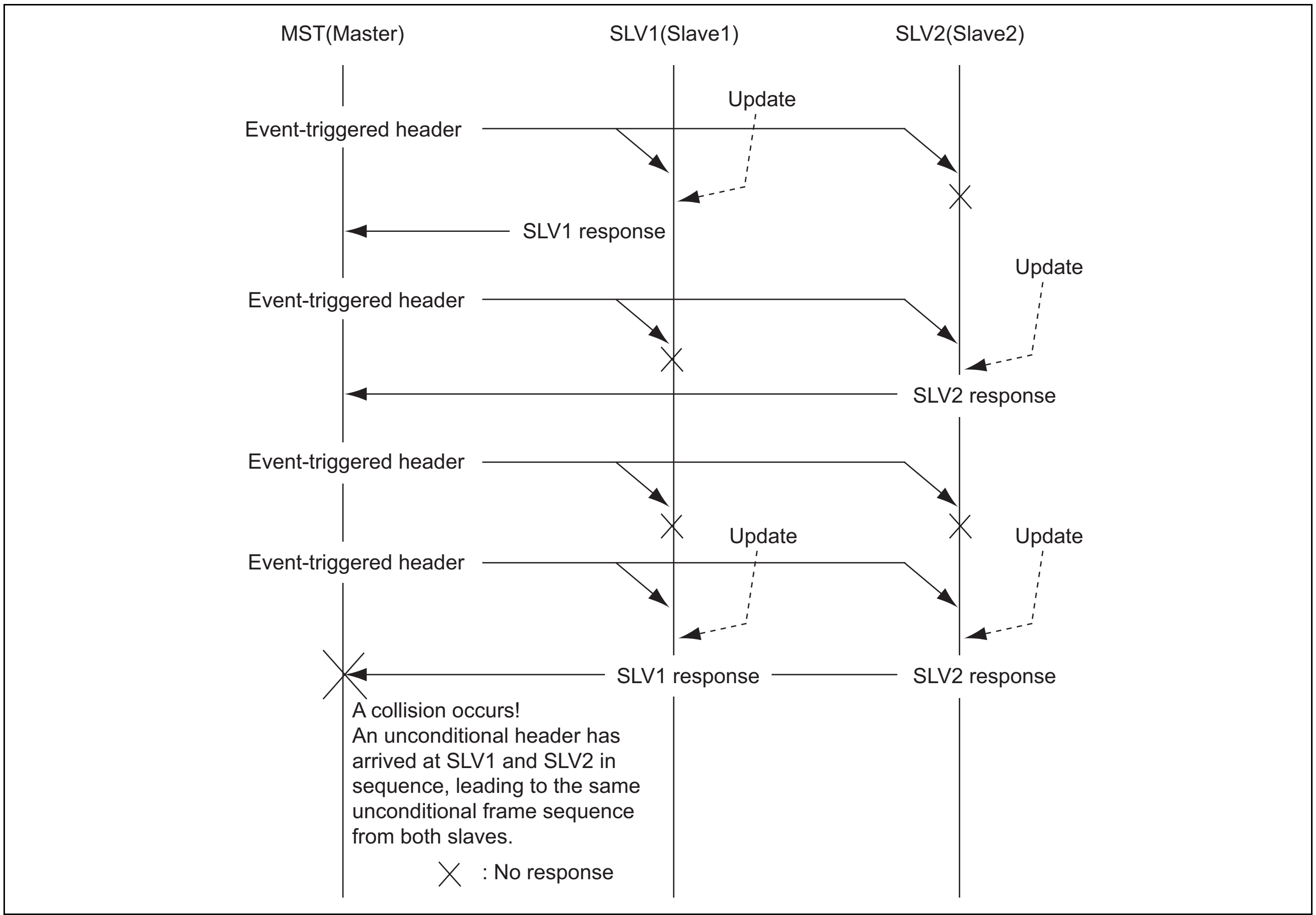

Figure 3 Sequences for event-triggered frames

1.2.3 Sporadic Frame

Sporadic frames are used to inform all relevant slave nodes of the updating of a signal value managed by the master node. Only the master node sends out a response to the header.

The sequence for a sporadic frame is illustrated in figure 4.

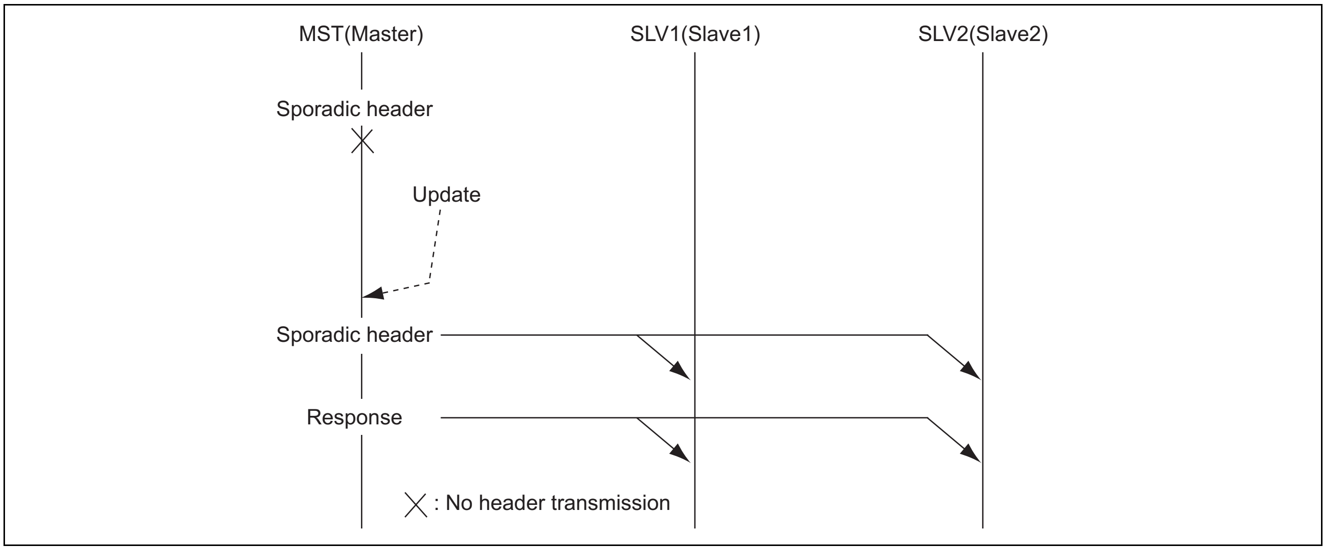

Figure 4 The sequence for a sporadic frame

1.2.4 Master Request Frame

Master request frames are used to transmit node settings and node-diagnostic information from the master node to slave nodes. Only the master node sends out a response to the header.

The sequence for a master request frame is illustrated in figure 5.

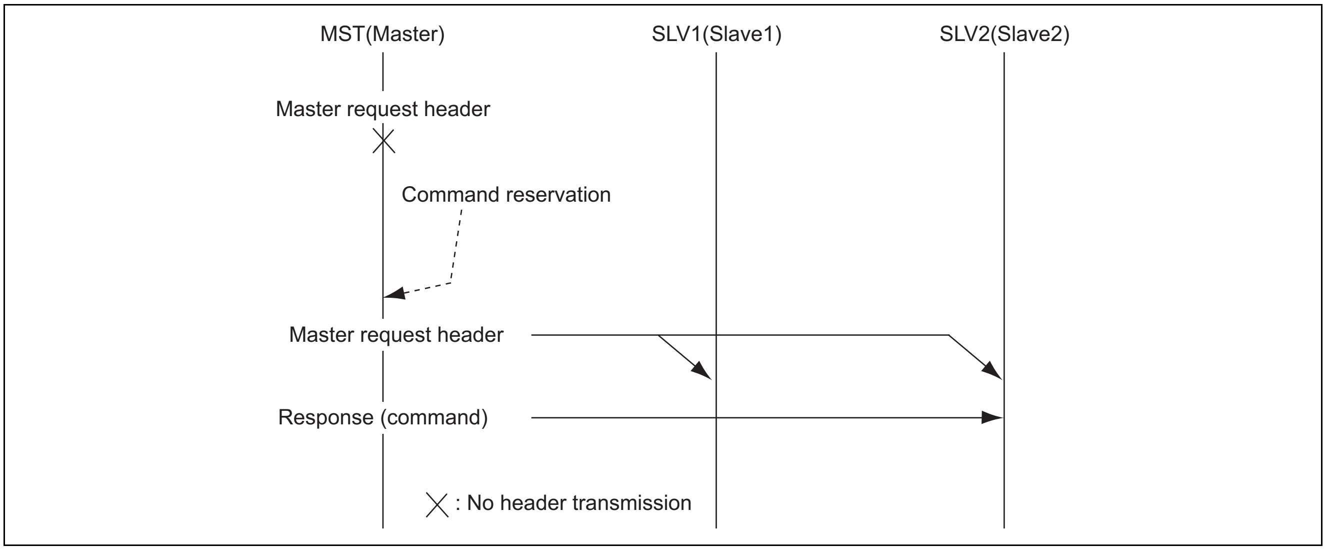

Figure 5 The sequence for a master request frame

1.2.5 Slave Response Frame

Slave response frames provide a way for the master node confirmations of validity or invalidity in response to node-diagnostic frames and responses to node-setting frames sent from the master node to the slave node. Only slave nodes send out responses to the header. This flow should not be implemented in the clustered structures where several slave nodes might react. Slave nodes will not transmit a response when they have nothing with which to respond.

The sequence for a slave response frame is illustrated in figure 6.

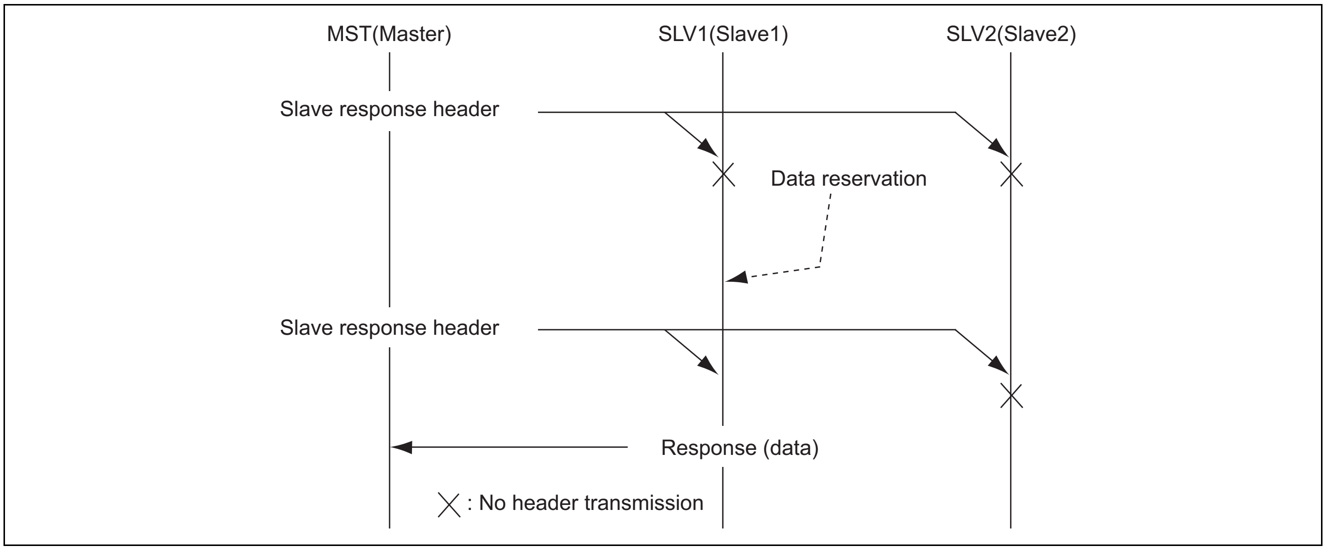

Figure 6 The sequence for a slave response frame

2. Specifications of LIN2.0 Library

Including the library in a user application program allows the program to use the on-chip functions of the H8/36049F to perform LIN communications as a master node.

2.1 Configuration of Files for the Library

- 36049s.h (Ver.1.00)
  This file contains definitions of the on-chip I/O registers for the H8/36049F

- sci_drv36049.c (Ver.1.00)
  This is the C source file for the driver that sets up and controls the SCI3 module to handle communications by the H8/36049F as a LIN master node. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with user application program at compile time for embedding in systems that employ LIN communications.

- sci_drv36049.h (Ver.1.00)
  This is the C header file for the driver that sets up the SCI3 module to handle communications by the H8/36049F as a LIN master node and controls LIN communications. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with the user application program at compile time for embedding in systems that employ LIN communications.

- tmr_drv36049.c (Ver.1.00)
  This is the C source file for the driver that sets up and controls counting by the timer B module to handle communications by the H8/36049F as a LIN master node. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with the user application program at compile time for embedding in systems that employ LIN communications.

- tmr_drv36049.h (Ver.1.00)
  This is the header file for the driver that sets up and controls counting by the timer B module to handle communications by the H8/36049F as a LIN master node. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with the user application program at compile time for embedding in systems that employ LIN communications.

- Lin_Drv36049.c (Ver.1.00)
  This is the C source file for the LIN driver that actually sets up and controls communications by the H8/36049F as a LIN master node. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with the user application program at compile time for embedding in systems that employ LIN communications.

- Lin_Drv36049.h (Ver.1.00)
  This is the header file for the LIN driver that actually sets up and controls communications by the H8/36049F as a LIN master node. This file can be freely modified or converted to operate with the CPU environment being employed by the user. Since the functions of this file are not included in the LIN2.0 library, it must be included with the user application program at compile time for embedding in systems that employ LIN communications.

- Lin_Master_Cnf.c (Ver.1.00)
  This file contains definitions specific to master nodes, and covers the handling of signals, frames, scheduling, and other items within clusters. Although this file is employed in the creation of cluster environments by the user, it is generally created by using the configurator.

- Lin_Com_Cnf.h (Ver.1.00)
  This header file is used to include the master-node definition file (Lin_Master_Cnf.c).

- lin20.h (Ver.1.00)
  This is the header file for the LIN2.0 library. This file must be included in the user programs for applications.

- lin20.lib (Ver.1.00)
  This is the main body of the LIN2.0 library. This file must be linked with the user programs for applications that employ LIN communications.

2.2 ROM/RAM Capacity

(The compiler in use is version V.6.00.03.000 of the C/C ++ compiler for the H8S Family and H8/300 Series.)

Amount of ROM/RAM given in this application note are amounts used by the LIN2.0 library (lin20.lib) alone, and otherwise will vary with other functions.

- ROM: 13356 bytes
- RAM: 234 bytes *

*: This does not include the heap requirements. Refer to Heap Area in section 2.2.1. below.

2.2.1 Heap Area

The buffers for the LIN2.0 library are dynamically allocated from the heap during initialization. Therefore, the development of applications that employ the library requires that a sufficiently large unused part of the heap be available. The following items indicate the minimum requirements for the heap area. Also, the items indicate how much memory from the heap will be required.

1. Minimum requirements for the heap (RAM) area
   — RAM buffer for controlling interface: 20 bytes
   — FIFO buffer for transmitting a frame of raw diagnostic data: 9 bytes (when one stage is saved.)
   — FIFO buffer for receiving a frame of raw diagnostic data: 9 bytes (when one stage is saved.)
   The above items require no less than 18 bytes of the heap.

2. Items that consume the heap area
   — FIFO buffers for transmitting frames of raw diagnostic data
   — FIFO buffers for receiving frames of raw diagnostic data
   The user can specify the number of stages of FIFOs listed above by using the configurator. For both transmission and reception, any number of stages from 1 to 65535 is specifiable.
   Calculation of heap area where memory is consumed is as follows:
      Formula for calculation: No. of stages of FIFO for transmission (or reception) x 9 bytes (amount required per stage of the FIFO)
   Example: when saving 30 stages of FIFO buffer for transmission and 20 stages of FIFO buffer for reception,
      (30 (stages) x 9 (bytes) + 20 (stages) x 9 (bytes) = 450 (bytes)

Note: When the required heap area is not available, an error occurs in the initialization of LIN system.

2.3 API Function

Functions of the LIN2.0 library for use by master nodes are described in this section. The style used to describe the API function is shown in figure 7.

Overview of function is indicated here.

Type of library function (return value and arguments) is indicated here.

Description	Describes the purpose of the library function.
Return value	Normal: the value or values returned when the library function ends normally. Abnormal: the value or values returned when the library function ends abnormally.
Argument	Describes the meaning of the arguments.
Example	Describes the procedure used to call the function.
Note	Supplementary descriptions or precautions

Figure 7 Style of descriptions of API functions

2.3.1 List of API Function

Table 2 is a list of API functions (a total of 36 functions) that master nodes can use.

Table 2 List of API functions

Name of API Function	Usage
l_sys_init	Initializes the LIN system
l_ifc_init	Initializes the interface
l_ifc_ioctl	Registers an I/O driver
l_ifc_connect	Makes a connection with the LIN bus
l_ifc_disconnect	Breaks a connection with the LIN bus
l_sch_set	Sets a schedule
l_sch_tick	Executes a schedule
l_flg_tst	Tests a flag
l_flg_clr	Clears a flag
l_bool_rd	Reads a 1-bit signal
l_u8_rd	Reads a 2- to 8-bit signal
l_u16_rd	Reads a 9- to 16-bit signal
l_bytes_rd	Reads byte assignment signals
l_bool_wr	Writes a 1-bit signal
l_u8_wr	Writes a 2- to 8-bit signals
l_u16_wr	Writes a 9- to 16-bit signals
l_bytes_wr	Writes data for a byte-assignment signal
l_ifc_goto_sleep	Reserves a sleep command
l_ifc_wake_up	Outputs a wake-up signal
l_ifc_tx	Transmits one frame
l_ifc_rx	Receives one frame
l_ifc_read_status	Acquires state information
ld_is_ready	Verifies state information on node setting
ld_check_response	Acquires the state information on response
ld_assign_frame_id	Assigns the frame ID
ld_read_by_id	Reads node property
ld_assign_NAD	Assigns NAD value
ld_conditional_change_NAD	Assigns conditional NAD value
ld_put_raw	Transmits a frame of raw diagnostic data
ld_get_raw	Acquires a frame of diagnostic data
ld_raw_tx_status	Acquires the state of raw diagnostic data transmitted
ld_raw_rx_status	Acquires the state of raw diagnostic data received
ld_send_message	Transmits a frame of processed diagnostic data
ld_receive_message	Receives a frame of processed diagnostic data
ld_tx_status	Acquires the state information on processed diagnostic data transmitted
ld_rx_status	Acquires the state information on processed diagnostic data received

2.3.2 Core API

System Initialization

l_bool l_sys_init(void)

Description	Initializes the LIN system
Return value	Normal initialization: 0 Failure in initialization: 1
Argument	None
Example	l_bool ret ret = l_sys_init();
Note	Call this API function first, i.e. before calling any of the API functions described below. This function is called only once after a reset.

Interface Initialization

void l_ifc_init(l_u8 ifc_name)

Description	Initializes a LIN interface
Return value	None
Argument	ifc_name Name of the interface
Example	l ifc_init(0);
Note	Call functions l_sys_init and l_ifc_ioctl before calling this function. Until ifc_init is called, operation in response to calling any API function other than the above is undefined, The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

I/O Driver Registration

l_u16 l_ifc_ioctl(l_u8 ifc_name, l_ioctl_op op, void* hand)

Description	registers the I/O drivers used by the individual nodes
Return value	When all drivers are registered: 0 When some drivers have not been registered: Number of unregistered drivers
Argument	ifc_name Name of the interface op Operation code hand Pointer for handling of a registered driver

Example	const T_Lib_Master_Handle Master_handle = {
	Lin_Drv_Init,
	Lin_Drv_BreakOut,
	Lin_Drv_BreakFinish,
	Lin_Drv_SendSync,
	Lin_Drv_SendPid,
	Lin_Drv_SendPidFinish,
	Lin_Drv_First_SendData,
	Lin_Drv_SendData,
	Lin_Drv_First_RecvReq,
	Lin_Drv_RecvData,
	Lin_Drv_SendRecvFinish,
	Lin_Drv_LinBus_Enable,
	Lin_Drv_LinBus_Disable,
	Lin_Drv_WakeUp,
	Lin_Drv_WakeUpFinish
	};
	l_u16 ret;
	ret = l_ifc_ioctl(0, LIN_ENTRY_MASTER_DRV, &Master_handle);

Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. Specify either of the following two codes as the operation code. Registration of the master-node drivers: LIN_ENTRY_MASTER_DRV Registration of the slave-node drivers: LIN_ENTRY_SLAVE_DRV Call this API function before calling the API function l_ifc_init.

LIN Bus Connection

l_bool l_ifc_connect(l_u8 ifc_name)

Description	Makes a connection with the LIN bus
Return value	Successful connection: 0 Failure to connect: 1
Argument	ifc_name Name of the interface

Example	l_bool ret;
	ret = l_ifc_connect(0);
	if(ret) {
	    /* Lin bus Connection failed */
	}

Note	Perform scheduled execution for LIN communications after calling this function to connect the device with the LIN bus. The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

LIN Bus Disconnection

l_bool l_ifc_disconnect(l_u8 ifc_name)

Description	Breaks a connection with the LIN bus
Return value	Successful disconnection: 0 Failure to disconnect: 1
Argument	ifc_name Name of the interface

Example	l_bool ret;
	ret = l_ifc_connect(0);
	if(ret) {
	    /* Lin bus disconnection failed */
	}

Note	When ending a session of LIN communications, disconnect the device from the LIN bus by calling this function. The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

Schedule Setting

void l_sch_set(l_u8 ifc_name, l_schedule_handle schedule, l_u8 entry)

Description	Sets a schedule table for execution from the next round of scheduled execution
Return value	None
Argument	ifc_name Name of the interface schedule Name of the schedule table entry Entry No.
Example	l_sch_set (0, &Lin_Sch_Schedule1, 3);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. The name of the schedule table is defined by the LIN configurator. The LIN configurator generates the schedule entity (table) defined by the user. The entity is placed within an output file. Here, add "&" to the name of the defined schedule entity to set the address of the entity. Ensure that this does not result in the setting of the null pointer or any other undefined address. The entry number indicates the order of the entry from which the set schedule is to be executed. Set a number within the number of entries in the set schedule table. Operation is not guaranteed in cases where a value outside this range is set. Both 0 and 1 are valid specifications, however, and indicate that execution should proceed from the first entry in the schedule. Refer to the documents for reference listed in section 3 for details.

Schedule Execution

l_u8 l_sch_tick(l_u8 ifc_name)

Description	Executes a schedule for one slot.
Return value	When a schedule is being executed: Number of the next entry to be executed When frame transmission has not been executed and 1 slot only was occupied: 0
Argument	ifc_name Name of the interface

Example	l_u8 entry;
	entry = l_sch_tick (0);
	if(entry) {
	/* Something is done in response to the entry. */
	}

Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. This API function should be called at the time-base interval set by the LIN configurator. Operation is not guaranteed when the function is not called at this interval. For details, refer to the documents listed in section 3, References.

Flag Test

l_bool l_flg_tst(l_flag_handle flag_name)

Description	Tests a flag
Return value	Value of the flag: 0 or 1
Argument	flag_name Name of the flag

Example	l_bool ret;
	ret = l_flg_tst(&Lin_Frm_FrameU1_flg);
	if(ret) {
	/* Something is done. */
	}
	else {
	/* Something else is done. */
	}

Note	The name of the interface can only be set to 0. In other words, it should not be set to any other value other than 0. The name of the flag is a name defined by the user. The address defined for the flag is substituted for this.

Flag Clearing

l_bool l_flg_tst(l_flag_handle flag_name)

Description	Clears a flag
Return value	None
Argument	flag_name Name of the flag
Example	l_flg_clr(&Lin_Frm_FrameU1_flg);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. The name of the flag is a name defined by the user. The address defined for the flag is substituted for this.

Signal Value Reading

l_bool l_bool_rd(l_signal_handle sig_name)

Description	Reads a 1-bit signal
Return value	Value of signal : 0 or 1
Argument	sig_name Name of the signal
Example	l_bool value; value = l_bool_rd(& Lin_Sig_Test0);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 1-bit signal. Operation is not guaranteed when the function is called to read such data.

Signal Value Reading

l_u8 l_u8_rd(l_signal_handle sig_name)

Description	Reads a 2- to 8-bit signal
Return value	Value of signal : 0 to 255
Argument	sig_name Name of signal
Example	l_u8 value; value = l_u8_rd(&Lin_Sig_Test3);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 2- to 8-bit signal. Operation is not guaranteed when the function is called to read such data.

Signal Value Reading

l_u16 l_u16_rd(l_signal_handle sig_name)

Description	Reads a 9- to 16-bit signal
Return value	Value of the signal : 0 to 65535
Argument	sig_name Name of signal
Example	l_u16 value; value = l_u16_rd(&Lin_Sig_Test7);
Note	The name of the flag is a defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 9- to 16-bit signal, using this API function. Operation is not guaranteed when the function is called to read such data.

Signal Value Reading

void l_bytes_rd(l_signal_handle sig_name, l_u8 start, l_u8 count, l_u8* const data)

Description	Reads data to a byte-assignment signal
Return value	None
Argument	sig_name Name of the signal start Location of the byte where writing is to start count Number of bytes to be read data buffer for holding the signal value: 1 to 8 bytes
Example	l_u8 data[8]; l_bytes_rd(&Lin_Sig_Test13, 1, 2);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a byte-assignment signal. Operation is not guaranteed when the function is called to read such data. Also, do not set a number of bytes that extends the defined signal size. Reading does not proceed if an error occurs. The content of buffer then is undefined.

Signal Value Writing

void l_bool_wr(l_signal_handle sig_name, l_bool sig)

Description	Writes a 1-bit signal
Return value	None
Argument	sig_name Name of the signal sig Value of signal: 0 or 1
Example	l_bytes_wr(&Lin_Sig_Test1, 1);
Note	The Name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 1-bit signal. Operation is not guaranteed when the function is called to read such data.

Signal Value Writing

void l_u8_wr(l_signal_handle sig_name, l_u8 sig)

Description	Writes a 2- to 8-bit signal
Return value	None
Argument	sig_name Name of the signal sig Value of signal: 0 to 255
Example	l_u8_wr(&Lin_Sig_Test4, 123);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 2- to 8-bit signal. Operation is not guaranteed when the function is called to read such data.

Signal Value Writing

void l_u16_wr(l_signal_handle sig_name, l_u16 sig)

Description	Writes a 9- to 16-bit signal
Return value	None
Argument	sig_name Name of the signal sig Value of signal: 0 to 65535
Example	l_u16_wr(&Lin_Sig_Test4, 12345);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 9- to 16-bit signal. Operation is not guaranteed when the function is called to read such data.

Signal Value Writing

void l_bytes_wr(l_signal_handle sig_name, l_u8 start, l_u8 count, const l_u8* const data)

Description	Writes a byte-assignment signal
Return value	None
Argument	sig_name Name of the signal start Location of the byte where reading is to start count Number of bytes to be read data buffer for holding the signal value: 1 to 8 bytes
Example	l_u8 data[8] = { 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0 }; l_bytes_wr(&Lin_Sig_Test15, 0, 8);
Note	The name of the flag is a name defined by the user. The address defined for the flag is substituted for this. Do not call this function to read a signal which is not actually a 9- to 16-bit signal. Operation is not guaranteed when the function is called to read such data. Also, do not set a number of bytes that extends signal size. Reading does not proceed if an error occurs. The content of buffer then is undefined.

Sleep Command

void l_ifc_goto_sleep(l_u8 ifc_name)

Description	Reserves the execution of a sleep command
Return value	None
Argument	ifc_name Name of the interface
Example	l_ifc_goto_sleep(0);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. The sleep command is not transmitted as soon as this function is called. Instead, transmission is in response to the next master request frame. Even when the transmission of other frames has been reserved, this command will take priority. At that time, such frames will be transmitted in response to the next master request frame. In cases of consecutive calls of this API function, the second and later calls are ignored.

Wake-Up Signal

void l_ifc_wake_up(l_u8 ifc_name)

Description	Outputs a wake-up signal
Return value	None
Argument	ifc_name Name of the interface
Example	l_ifc_wake_up(0);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. The wake-up signal is output when this API function is called.

Frame Transmission

void l_ifc_tx(l_u8 ifc_name)

Description	Transmits a frame
Return value	None
Argument	ifc_name Name of the interface

Example	vodi tx_isr(void)
	{
	l_ifc_tx(0);
	}

Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. This API function is normally called within a handler for interrupt-driven serial transmission. The location of the call will depend on the configuration of the hardware.

Frame Reception

void l_ifc_rx(l_u8 ifc_name)

Description	Receives a frame
Return value	None
Argument	ifc_name Name of the interface

Example	vodi rx_isr(void)
	{
	    l_ifc_rx(0);
	}

Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. This API function is normally called within a handler for interrupt-driven serial transmission. The location of the call will depend on the configuration of the hardware.

State-information Acquisition

l_u16 l_ifc_read_status(l_u8 ifc_name)

Description	Status value: See section 3, Refe rences.
Return value	Successful disconnection: 0 Failure to disconnect: 1
Argument	ifc_name Name of the interface
Example	l_u16 status; status = l_ifc_read_status(0);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

2.3.3 API for Node Setting

Verification of Node Setting

l_bool ld_is_ready(l_u8 ifc_name)

Description	Verfies a state of node setting.
Return value	When the node setting API function is called: 1 When the node setting API function is not called: 0
Argument	ifc_name Name of the interface
Example	lif(ld_is_ready(0)) { };
Note	Name of interface can only be set to 0. In other words, it should not be set to any other numbers except for 0. When the slave response frame is registered to the schedule after the master request frame, this function is set after the slave response frame is executed. When the schedule is changed after the master request frame is transmitted and the slave response frame is not transmitted, this function is not set until the slave response frame is transmitted. Execute the master request frame and slave response frame continuously (recommended).

Verification of Response

l_u8 ld_check_response(l_u8 ifc_name, l_u8* rsid, l_u8* error_code)

Description	Verifies the state of response
Return value	State of response: See section 3, References.
Argument	ifc_name Name of the interface rsid buffer for saving the response ID error_code buffer for saving the error code
Example	l_u8 rtn, rsid, error_code; rtn = ld_check_response(0, &rsid, &error_code);
Note	Name of interface can be set only to 0. In other words, it should not be set to any other numbers except for 0.

Frame ID Assignment

void ld_assign_frame_id(l_u8 ifc_name, l_u8 nad, l_u16 supplier_id, l_u16 message_id, l_u8 pid)

Description	Reserves execution of the command to assign a frame ID
Return value	None
Argument	ifc_name Name of the interface nad NAD value of the target node supplier_id Supplier ID of the target node message_id Message ID of the target node pid Protected frame ID corresponding to the frame ID being assigned
Example	ld_assign_frame_id(0, 0x23u, 0x1234u, 0x4567u, 0x61u);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. Frame ID Assignment is not transmitted as soon as this function is called. Instead, transmission is in response to the next master request frame. When the execution of the sleep command is reserved then, the sleep command is prioritized. Since there is no return value, error checking is not automatically executed. However, the checking should be executed on the side that calls this function.

Node Property Reading

void ld_read_by_id(l_u8 ifc_name, l_u8 nad, l_u16 supplier_id, l_u16 function_id, l_u8 id, l_u8* const data)

Description	Reserves execution of the command for reading node properties.
Return value	None
Argument	ifc_name Name of the interface nad NAD value of the target node supplier_id Supplier ID of the target node function_id Function ID of the target node data Buffer for saving the data read from the node
Example	l_u8 data[8]; ld_read_by_id(0, 0x23, 0x1234, 0x6789, 1, data);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. When this API function is called, transmission is not executed. Node property reading is not transmitted as soon as this function is called. Instead, transmission is in response to the next master request frame. When the execution of the sleep command is reserved then, the sleep command is prioritized. Note that the buffer for saving the data must have 8 bytes. Since there is no return value, error checking is not automatically executed. However, the checking should be executed on the side that calls this function.

NAD Value Assignment

void ld_assign_NAD(l_u8 ifc_name, l_u8 nad, l_u16 supplier_id, l_u16 function_id, l_u8 new_NAD)

Description	Reserves the execution of the command to assign a new NAD value
Return value	None
Argument	ifc_name Name of the interface nad Current NAD value of the target node supplier_id Supplier ID of the target node function_id Function ID of the target node new_NAD New NAD value
Example	ld_assign_NAD(0, 0x23, 0x1234, 0x5678, 0x15);
Note	Name of interface can be set only to 0. In other words, it should not be set to any other numbers except for 0. When this API function is called, transmission is not executed. NAD value assignment is not transmitted as soon as this function is called. Instead, transmission is in response to the next master request frame. When the execution of the sleep command is reserved then, the sleep command is prioritized. Since there is no return value, error checking is not automatically executed. However, the checking should be executed on the side that calls this function.

Conditional NAD Value Change

void ld_conditional_change_NAD(l_u8 ifc_name, l_u8 nad, l_u8 id, l_u8 byte, l_u8 mask, l_u8 invert, l_u8 new_NAD)

Description	Reserves the execution of the command for conditionally assigning a new NAD value
Return value	None
Argument	ifc_name Name of the interface nad Current NAD value of the target node id Property ID of the target node byte Byte location of property value to be read from the target node mask Value for masking the read property byte invert Value for excluding the read property byte new_NAD New NAD value to be assigned when the condition is met
Example	ld_conditional_change_NAD(0, 0x23, 1, 2, 0x55, 0xAA, 0x15);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. When this API function is called, transmission is not executed. Conditional NAD Value Change is not transmitted as soon as this function is called. Instead, transmission is in response to the next master request frame. When the execution of the sleep command is reserved then, the sleep command is prioritized. Since there is no return value, error checking is not automatically executed. However, the checking should be executed on the side that calls this function.

2.3.4 API for Frames of Diagnostic Data

Reservation of the Transmission for a Frame of Raw Diagnostic Data

void ld_put_raw(l_u8 ifc_name, const l_u8* const data)

Description	Reserves the transmission of a frame of raw diagnostic data from the transmission FIFO buffer
Return value	None
Argument	ifc_name Name of the interface data Buffer for the data to be transmitted
Example	l_u8 data[8] = { 0x20u, 0x06u, 0xb1u, 0xffu, 0x7fu, 0x00u, 0x00u, 0x20u }; ld_put_raw(0, data);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. The transmission does not proceed as soon as the API function is called. Instead, transmission is in response to the next master request frame. At that time, however, a sleep command or node-setting command for which execution has also been reserved will take priority over this command. When the required space is not available in the FIFO buffer, execution of the command is not reserved in response to the function call. Since there is no return value, the function itself does not cover error checking. However, checking should be executed on the calling side.

Acquisition of a Frame of Raw Diagnostic Data

void ld_get_raw(l_u8 ifc_name, l_u8* const data)

Description	Acquires a frame of raw diagnostic data from the FIFO buffer
Return value	None
Argument	ifc_name Name of the interface data Buffer for saving the acquired data
Example	l_u8 data[8]; ld_get_raw (0, data);
Note	The name of the interface can be only set to 0. In other words, it should not be set to any value other than 0. When this API function is called, the oldest frame of data is acquired from the FIFO buffer. Once the FIFO buffer is empty, no data is acquired even if this function is called. Since there is no return value, the function itself does not cover error checking. However, checking should be executed on the calling side.

Verification of the Transmission of a Frame of Raw Diagnostic Data

l_u8 ld_tx_status(l_u8 ifc_name)

Description	Verifies the state of transmission FIFO buffer in preparation for the transmission of a frame of raw diagnostic data
Return value	No available space in the FIFO buffer: LD_QUEUE_FULL FIFO buffer empty: LD_QUEUE_EMPTY An error in transfer has occurred: LD_TRANSFER_ERROR
Argument	ifc_name Name of the interface
Example	l_u8 rtn; rtn = ld_raw_tx_status (0);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

Transmission of a Frame of Processed Diagnostic Data

void ld_send_message(l_u8 ifc_name, l_u16 length, l_u8 NAD, const l_u8* const data)

Description	Reserves the transmission of a frame of processed diagnostic data
Return value	None
Argument	ifc_name Name of the interface length Amount of data for transmission NAD NAD value of the destination node for the transmission data Buffer for the data to be transmitted
Example	l_u8 data[5] = { 0x12, 0x34, 0x56, 0x78, 0x9A }; ld_send_message (0, 5, 0x23, data);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. Since there is no return value, error checking is not automatically executed. However, checking should be executed on the side that calls this function. If this function is called again before the transmission of the current frame is complete, operation is not guaranteed.

Reception of a Frame of Processed Diagnostic Data

void ld_receive_message(l_u8 ifc_name, l_u16* length, l_u8* NAD, l_u8* const data)

Description	Reserves reception of a frame of processed diagnostic data
Return value	None
Argument	ifc_name Name of the interface ength Buffer for storing received data NAD NAD value of the source node for the transmission data Buffer for storing received data
Example	l_u8 data[100], nad; l_u16 length = 100; ld_receive_message (0, &length, &nad, data);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0. Set the length of the buffer to hold the received data after having saved the permissible amount of received data at the time of reservation. Since there is no return value, the function itself does not cover error checking. However, checking should be executed on the calling side. If this function is called again before reception of the current frame is complete, operation is not guaranteed.

Verification of the State of Transmission of a Frame of Processed Diagnostic Data

l_u8 ld_tx_status(l_u8 ifc_name)

Description	Verifies the state of transmission of a frame of processed diagnostic data
Return value	Transmission complete: LD_COMPLETED Reception in progress: LD_IN_PROGRESS Reception failed: LD_FAILED
Argument	ifc_name Name of the interface
Example	l_u8 rtn; rtn = ld_tx_status(0);
Note	Name of interface can be only set to 0. In other words, it should not be set to any value other than 0.

Verification of the State of Reception of a Frame of Processed Diagnostic Data

l_u8 ld_tx_status(l_u8 ifc_name)

Description	Verifies the state of reception of a frame of processed diagnostic data
Return value	Reception completed: LD_COMPLETED Reception in progress: LD_IN_PROGRESS Reception failed: LD_FAILED
Argument	ifc_name Name of the interface
Example	l_u8 rtn; rtn = ld_tx_status(0);
Note	The name of the interface can only be set to 0. In other words, it should not be set to any value other than 0.

2.4 How to Use the API Functions of the LIN Library

Examples of the usage of the API functions of the LIN2.0 library are given below.

2.4.1 Initialization of LIN System

The LIN system must be initialized before the API functions of the LIN2.0 library are used.

In the example below, the LIN system is initialized when the microcomputer is reset.

Note that this reflects the points where the API functions for LIN are called.

```
extern unsigned char lin_SomeCotrol_init( void );
__entry(vect=0) void PowerON_Reset(void)
{
    set_imask_ccr(1);

   _INITSCT();
// _CALL_INIT();             // Remove the comment to use global class object.
// _INIT_IOLIB();            // Remove the comment mark to use SIM I/O.
// errno=0;                  // Remove the comment mark to use errno.
// srand(1);                 // Remove the comment mark to use rand().
// _s1ptr=NULL;              // Remove the comment mark to use strtok().
   HardwareSetup();    // Remove the comment mark to use Hardware Setup.

   set_imask_ccr(0);

   /* ......Something to do */

   if( l_sys_init() ) {
      /* LIN System Initialization failed */
      sleep();
   }
   else {
      if( lin_SomeCotrol_init() ) {
         /* SomeSensor Initialization failed */
         sleep();
      }
   }

   /* ......Something to do */

   main();

// _CLOSEALL();              // Remove the comment mark to use SIM I/O.
// _CALL_END();              // Remove the comment mark to use global class
object.
   sleep();
}
```

```
/* Definitions for Master Driver Entry */
const T_Lib_Master_Handle Master_handle = {
   Lin_Drv_Init,
   Lin_Drv_BreakOut,
   Lin_Drv_BreakFinish,
   Lin_Drv_SendSync,
   Lin_Drv_SendPid,
   Lin_Drv_SendPidFinish,
   Lin_Drv_First_SendData,
   Lin_Drv_SendData,
   Lin_Drv_First_RecvReq,
   Lin_Drv_RecvData,
   Lin_Drv_SendRecvFinish,
   Lin_Drv_LinBus_Enable,
   Lin_Drv_LinBus_Disable,
   Lin_Drv_WakeUp,
   Lin_Drv_WakeUpFinish
};

/* Cluster Initialization */
extern T_Schedule Lin_Sch_Schedule1;     /* Schedule defined by the user */
unsigned char lin_SomeCotrol_init( void )
{
   unsigned char rtn;

   rtn = 0;
   if( l_ifc_ioctl( 0, LIN_ENTRY_MASTER_DRV, &Master_handle ) ) {
      /* The init of the LIN master driver failed */
      rtn = 1u;
   }
   else {
      l_ifc_init(0);   /* Interface Initialize */
      if( l_ifc_connect(0) ) {
         /* Connection of the LIN interface failed */
         rtn = 1u;
      }
      else {
         /* Schedule Setting */
         l_sch_set( 0, &Lin_Sch_Schedule1, 0 );
         lin_schedule_start();
      }
   }
   return rtn;
}

void lin_schedule_start( void )
{
   MSTCR1.BIT.MSTTW = 0;           /* Module standby canceled */
   TW.TCRW.BIT.CCLR = 0;           /* Free run */
   TW.TCRW.BIT.CKS = 3;            /* 8/ */
   TW.TIERW.BIT.OVIE = 1u;
   TW.TCNT = (0xFFFFu-2500u);
   TW.TMRW.BIT.CTS = 1u;           /* Start counting up */
}
```

2.4.2 Schedule Execution

In any LIN system, the API function for schedule execution must be called regularly. In the sample task below, timer W is used to count-up and generate interrupts at a 1-ms interval. Also, in the function (main processing) for schedule-table execution, the API function for schedule-table execution must be called at the corresponding time-base interval. In this sample task, the time base for schedule-table execution is defined as 500 ms.

Note that this reflects the points where the API functions for LIN are called.

```
static unsigned short tw_counter = 0;
/************************************/
/* 1-ms Interrupt Function for Timer W    */
/************************************/
__interrupt(vect=21)
void tw_isr_1ms( void )
{
   UB dummy;

   TW.TCNT = (0xFFFFu-2500u);
   dummy = TW.TSRW.BYTE;
   TW.TSRW.BYTE = 0;

   /* Something to do */

   tw_counter++;

   /* Something to do */

   return;
}

/************************************/
/* 1-ms Counter Acquisition Function    */
/************************************/
unsigned short lin_get_tw_counter( void )
{
   unsigned short c;

   /* Disable 1-ms timer interrupt */
   c = tw_counter;
   /* Enable 1-ms timer interrupt enable */
   return c;
}

/************************************/
/* 1-ms Counter Clear Function    */
/************************************/
void lin_clr_tw_counter( void )
{
   /* Disable timer w interrupt */
   TW.TIERW.BIT.OVIE = 0;

   tw_counter = 0;

   /* Enable timer w interrupt */
   TW.TIERW.BIT.OVIE = 1u;

   return;
}
```

```
/************************************/
/* LIN Schedule Function       */
/************************************/
void lin_schedule_exe( void )
{
   l_u8 entryno;

   if( LIN_TIME_BASE <= lin_get_tw_counter() ) {
      lin_clr_tw_counter();
      /* ......Something to do */

      entryno = l_sch_tick( 0 );

      /* ......Something to do */
   }

   return;
}
```

2.4.3 Applications

Sample codes regarding the API function in LIN2.0 library, which are called from other applications except for initialization and schedule execution, are described in this section. How to use the data acquired by calling the API function is dependent with different applications, so it is not specifically described in this application task. Contents (frame) transferred on the LIN bus is the data acquired from the status of various nodes, peripheral devices, and other applications. Therefore, types of data to be transferred or how to process data depends on LIN system configuration.

```
#include "36014s.h"
#include "Lin_Drv36014.h"
#include "lin20.h"
void lin_application( void );
/****************************/
/* Main Function        */
/****************************/
void main(void)
{
   while( 1 ) {
      /* ......Something to do */

      lin_application();

      /* ......Something to do */
   }
}
```

```
/*************************************/
/* LIN Application Function   */
/*************************************/
extern l_flg   Lin_Sig_Status_Slv1_flg;  /* Flag defined by the user */
extern T_Signal   Lin_Sig_Status_Slv0;   /* Signal defined by the user */
extern T_Signal   Lin_Sig_Command;       /* Signal defined by the user */
extern T_Schedule Lin_Sch_Schedule2;     /* Schedule defined by the user */
void lin_application( void )
{
   l_u16 signal;
   l_u8  rsid, error_code, ret_res;
   l_u8  data[8];
   union {
      l_u16 Word;
      struct {
         l_u16 lastpid      :8;
         l_u16         :4;
         l_u16 gotosleep    :1;
         l_u16 overrun      :1;
         l_u16 txsuccese    :1;
         l_u16 errorrsp     :1;
      } Bit;
   } status;

   /* Lin Schedule Cyclic Execution */
   lin_schedule_exe();

   if( l_flg_tst( &Lin_Sig_Status_Slv1_flg ) ) {     /* Verify the state of
flag */
      l_flg_clr( &Lin_Sig_Status_Slv1_flg );   /* Clear the state of flag  */
      signal = l_u16_rd( &Lin_Sig_Status_Slv0 );     /* Acquire the signal
value */
      l_u16_wr( &Lin_Sig_Command, signal );          /* Set the signal value
read */
   }

   /* Read status */
   status.Word = l_ifc_read_status( 0 );

   if( status.Bit.errorrsp ) {
      /* Something Error Response Processing   */
   }

   if( status.Bit.lastpid == 0x34u ) {
      l_sch_set( 0, &Lin_Sch_Schedule2, 2 );   /* Reset the execution schedule
*/
   }
```

```
    if( ld_is_ready( 0 ) ) {
        ret_res = ld_check_response( 0, &rsid, &error_code );
        switch( ret_res ) {
        case LD_NEGATIVE:
            /* Something is done */
            ld_read_by_id( 0, 0x23u, 0x1234u, 0x4321u, 0, data );
            break;
        case LD_SUCCESS:
            /* Something is done */
            break;
        case LD_NO_RESPONSE:
            /* Something is done */
            ld_assign_frame_id( 0, 0x20u, 0x1234u, 0x5678u, 0x61u );
            break;
        case LD_OVERWRITTEN:
            /* Something is done */
            break;
        default:
            /* Something is done */
            break;
        }
    }

    switch( ld_tx_status( 0 ) ) {
    case LD_COMPLETED:
        /* Something is done */
        break;
    case LD_IN_PROGRESS:
        /* Something is done */
        break;
    case LD_FAILED:
        /* Something is done */
        break;
    default:
        /* Something is done */
        break;
    }

    if( status.Bit.gotosleep ) {
        /* Something Sleep Mode Processing */
    }
}
```

3. References

- LIN Specification Package Revision 2.0: http://www.lin-subbus.org
- LIN Protocol Specification Revision 2.0: http://www.lin-subbus.org
- LIN Diagnostic and Configuration Specification Revision 2.0: http://www.lin-subbus.org
- LIN Application Program Interface Specification Revision 2.0: http://www.lin-subbus.org
- LIN Physical Layer Specification Revision 2.0: http://www.lin-subbus.org
- H8/36049 Group Hardware Manual: REJ09B0060-0200Z

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Jan.31.06	—	First edition issued