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

H8/300H Tiny Series H8/36014 Group

LIN (Local Interconnect Network): Slave Volume

Introduction

LIN (Local Interconnect Network): Slave Volume provides examples of settings and usage of the on-chip peripheral functions of H8/300H Tiny Series H8/36014 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 has been confirmed. However, be sure to confirm the operation before actual usage.

Target Device

H8/300H Tiny Series H8/36014F

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.

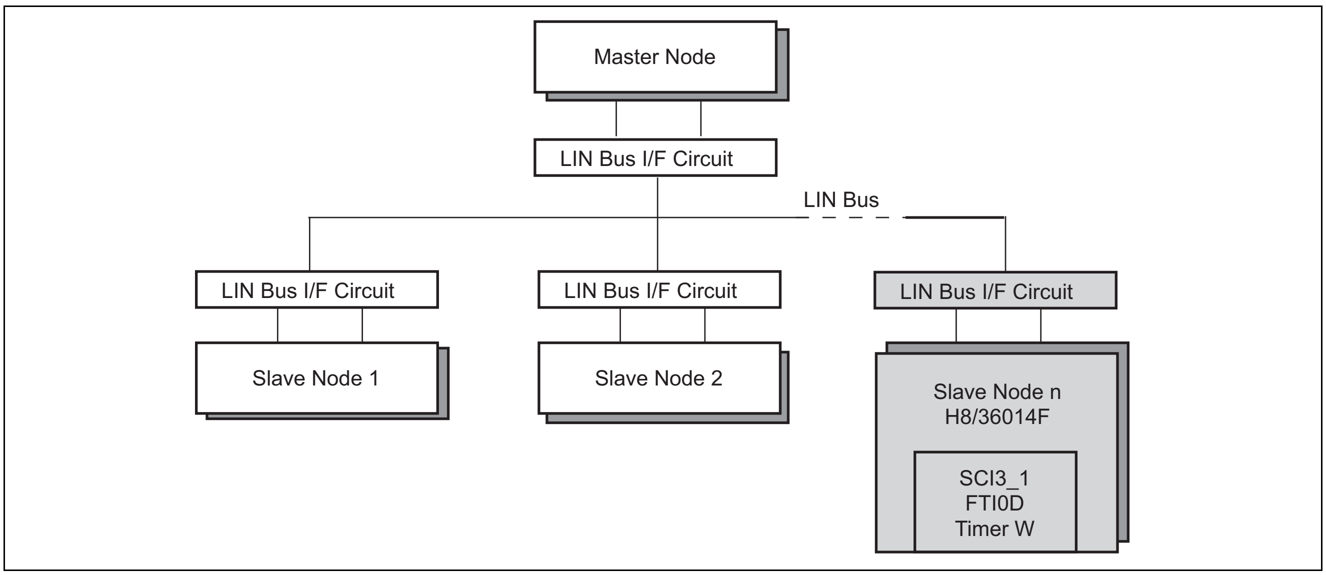

Figure 1 System configuration

1.1.2 Resource Usage

Resources of the H8/36014F for use in this application note are listed in table 1.

Table 1 CPU resources used in slave node operation

<table>
<tr><th colspan="2">Function</th><th>Pin Function (Pin No.)</th><th>Usage</th><th>Description/Comment</th></tr>
<tr><td colspan="2">I/O port pin</td><td>P80
(36)</td><td>LIN transceiver control</td><td>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.</td></tr>
<tr><td rowspan="3">SCI3
(Channel-1)</td><td>Transmission</td><td>TXD
(46)</td><td>Transmission of response frames, output of wake-up signal</td><td rowspan="2">Asynchronous mode, 8-bit data length, no parity bit, 1-stop bit (with start bit added), LSB first</td></tr>
<tr><td rowspan="2">Reception</td><td rowspan="2">RXD
(45)</td><td>Reception of response frames</td></tr>
<tr><td>Detection of errors in communications</td><td>Module’s internal error detection function</td></tr>
<tr><td colspan="2">Timer W</td><td>FTI0D
(40)</td><td>Header detection (input capture), measurement of wake-up signal period (output compare)</td><td>Automatic speed adjustment through detection of the break period and synchronization bytes.</td></tr>
</table>

1.2 Overview of LIN Communications

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.

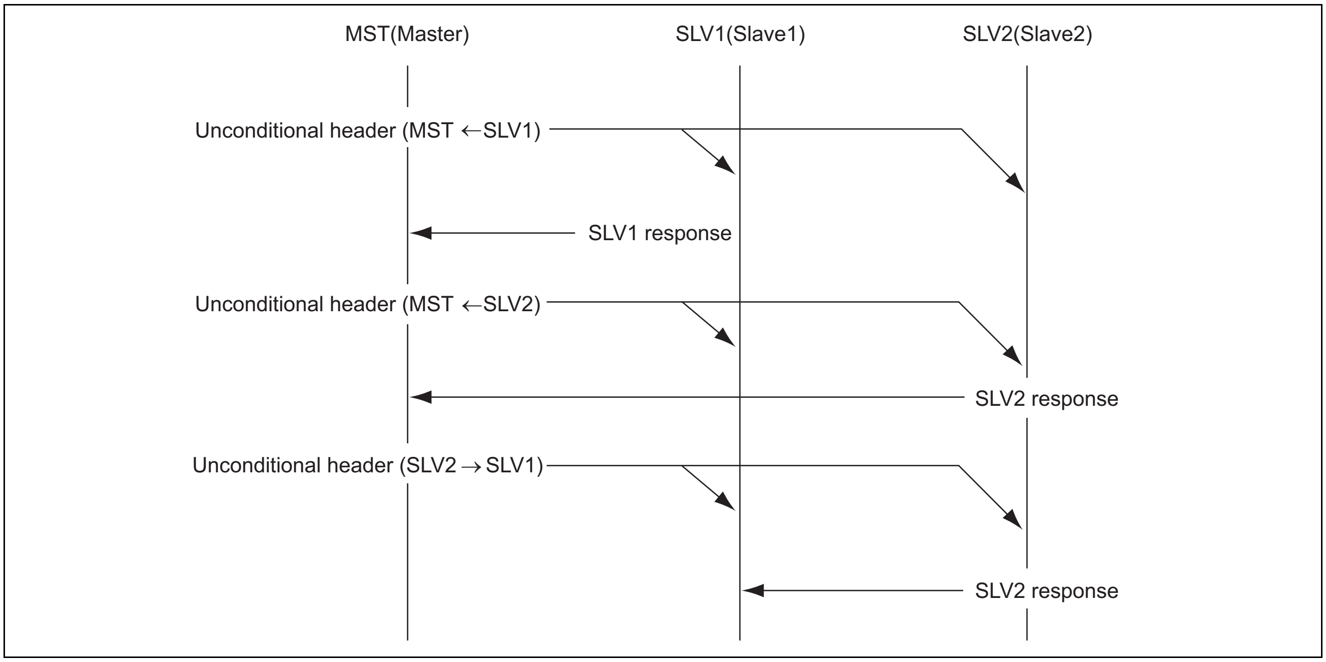

Figure 2 Sequences for unconditional frames

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.

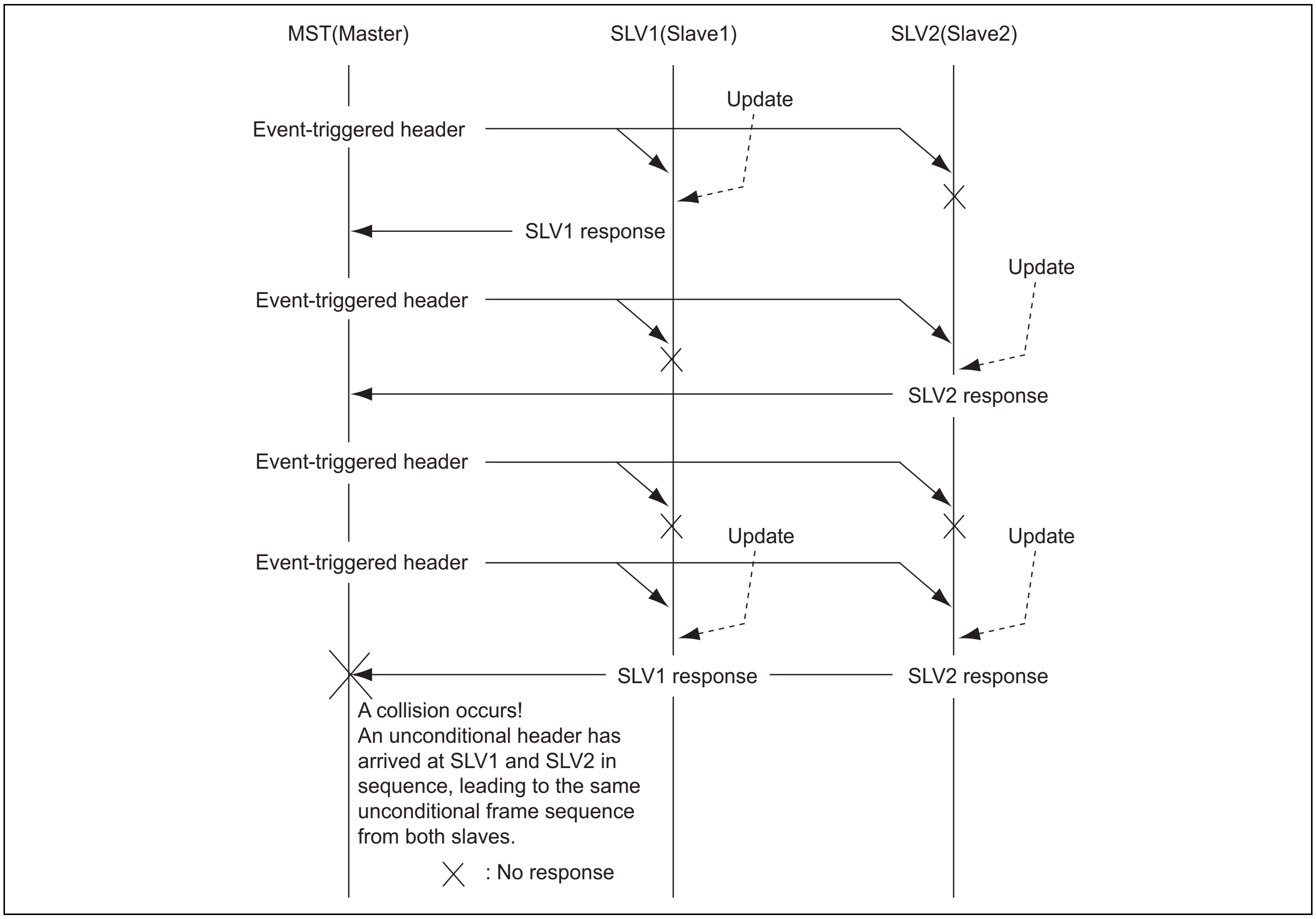

Figure 3 Sequences for event-triggered frames

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.

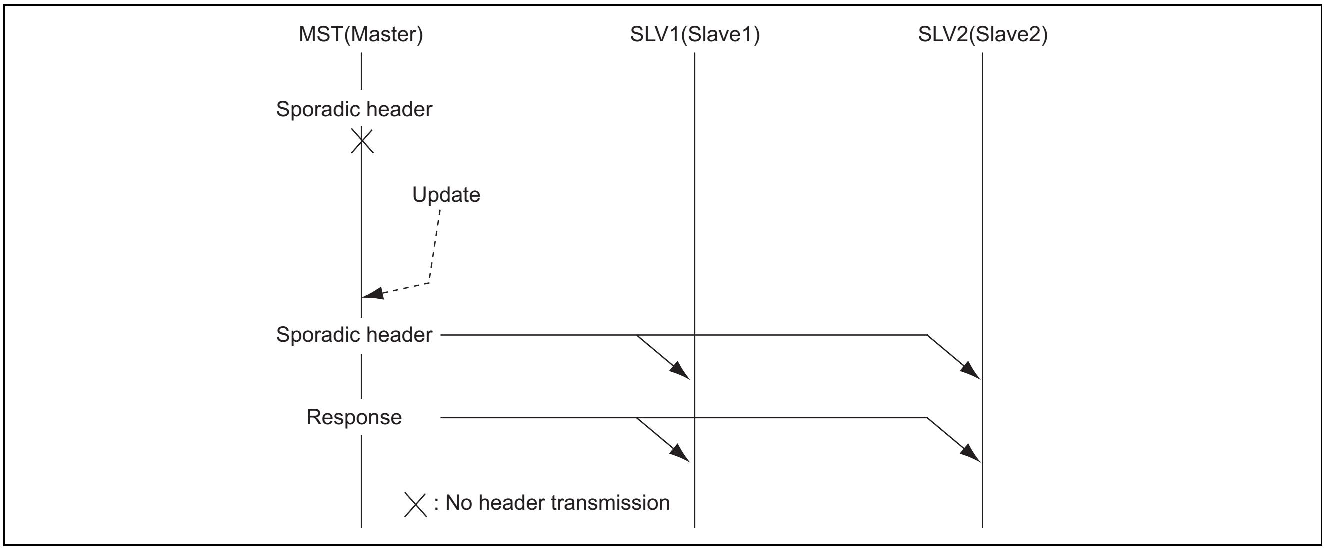

Figure 4 The sequence for a sporadic frame

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.

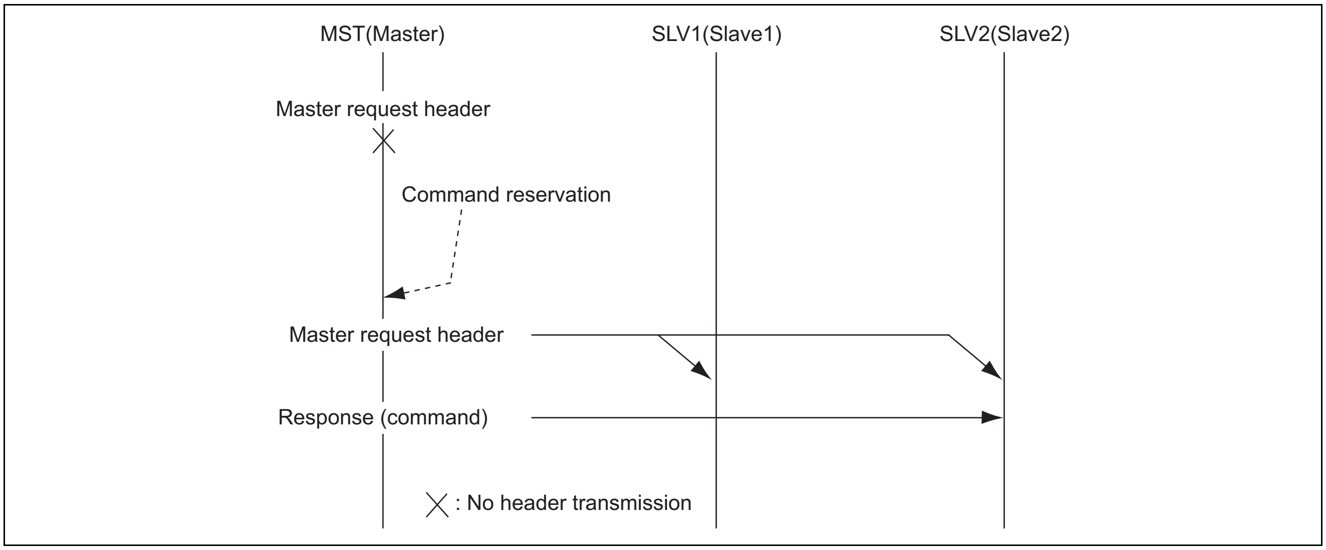

Figure 5 The sequence for a master request frame

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 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.

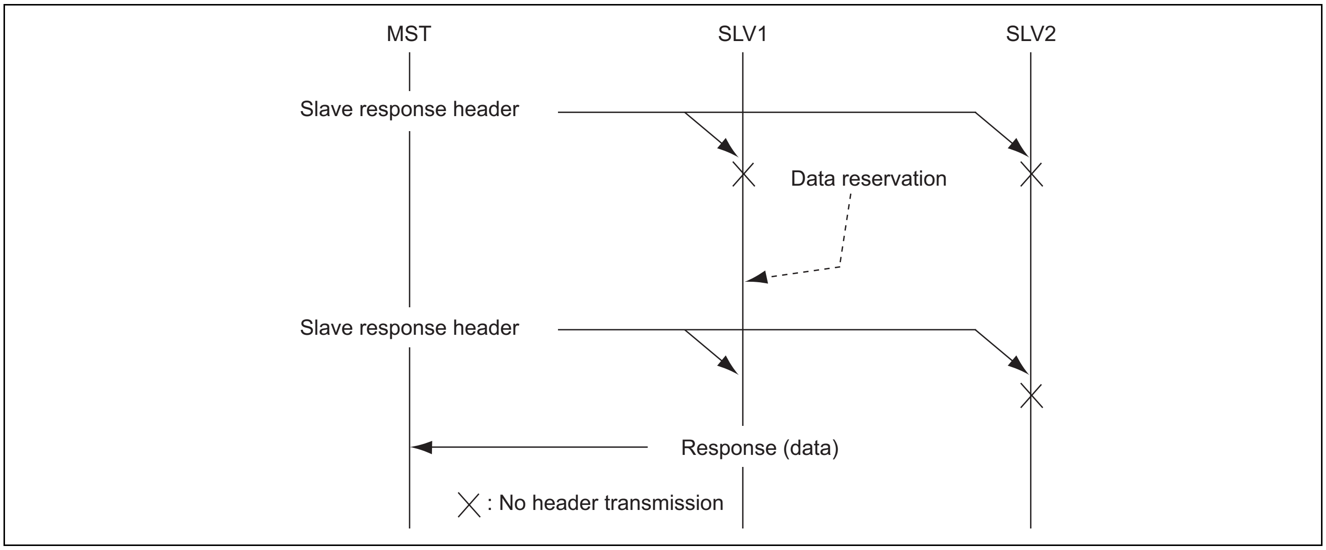

Figure 6 The sequence for a slave response frame

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 H8/36014F to perform LIN communications as a slave node.

2.1 Configuration of Files for the Library

- 36014s.h (Ver.1.00)
  This file contains definitions of the on-chip I/O registers for the H8/36014 Group.

- sci_drv36014.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 H8/36014F as a LIN slave 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.

- sci_drv36014.h (Ver.1.00)
  This is the header file for the driver that sets up the SCI3 module to handle communications by the H8/36014F as a LIN slave 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_drv36014.c (Ver.1.00)
  This is the C source file for the driver that sets up and controls counting by the timer W module to handle communications by the H8/36014F as a LIN slave 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_drv36014.h (Ver.1.00)
  This is the header file for the driver that sets up and controls counting by the timer W module to handle communications by the H8/36014F as a LIN slave 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_Drv36014.c (Ver.1.00)
  This is the C source file for the LIN driver that actually sets up and controls communications by the H8/36014F as a LIN slave 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_Drv36014.h (Ver.1.00)
  This is the header file for the LIN driver that actually sets up and controls communications by the H8/36014F as a LIN slave 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_Slave_Cnf.c (Ver.1.00)
  This file contains definitions specific to slave 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 slave-node definition file (Lin_Slave_Cnf.c).

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

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

2.2 ROM/RAM Capacity

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

Amounts 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 definitions.

- ROM: 8908 bytes
- RAM: 155 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
   — FIFO buffers for transmitting a frame of raw diagnostic data: 9 bytes (when one stage is saved.)
   — FIFO buffers 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 the FIFOs listed above by using the configurator. For both transmission and reception, any number of stages from 1 to 65535 is specifiable.
   The requirement for memory from the heap is calculated by using the following formula.
   Formula for calculation: No. of stages in the FIFO for transmission (or reception) x 9 bytes

   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 Functions

Functions of the LIN2.0 library for use by slave nodes are described in this section. The style used to describe the API functions 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 Functions

Table 2 is a list of the API functions (a total of 28 functions) that slave nodes can use.

Table 2 List of API functions

Name of 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_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 data out from a byte-array signal
l_bool_wr	Writes a 1-bit signal
l_u8_wr	Writes a 2- to 8-bit signal
l_u16_wr	Writes a 9- to 16-bit signal
l_bytes_wr	Writes data for a byte-array signal
l_ifc_wake_up	Outputs a wake-up signal
l_ifc_tx	Transmits one frame
l_ifc_rx	Receives one frame
l_ifc_aux	Detects Break/Synch. header
l_ifc_read_status	Acquires state information
ld_put_raw	Transmits a frame of raw diagnostic data
ld_get_raw	Acquires a frame of raw diagnostic data
ld_raw_tx_status	Acquires state information on the transmission of raw diagnostic data
ld_raw_rx_status	Acquires state information on the reception of raw diagnostic data
ld_send_message	Transmits a frame of processed diagnostic data
ld_receive_message	Receives a frame of processed diagnostic data
ld_tx_status	Acquires state information on the buffer for the transmission of raw diagnostic data
ld_rx_status	Acquires state information on the buffer for the reception of raw diagnostic data

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	ifc_init(0);
Note	Call functions l_sys_init and l_ifc_ioctl before calling this function. Until l_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.

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_Slave_Handle Slave_handle = {
	Lin_Drv_Init,
	Lin_Drv_BreakIn,
	Lin_Drv_BreakInFinish,
	Lin_Drv_BitInStart,
	Lin_Drv_BitIn,
	Lin_Drv_BitInEnd,
	Lin_Drv_SyncInFinish,
	Lin_Drv_Pid_RecvReq,
	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_SLAVE _DRV, &Slave_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 driver: 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.

Flag Testing

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 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 the 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 the signal: 0 to 255
Argument	sig_name Name of the 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 the signal
Example	l_u16 value; value = l_u16_rd(&Lin_Sig_Test7);
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 Reading

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

Description	Reads data out from a byte-array 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-array signal. Operation is not guaranteed when the function is called to read such data. Also, do not set a number of bytes that extends beyond the end of the signal. Reading out does not proceed if an error occurs, and in this case the contents of the buffer are 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 thesignal 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 the signal: 0 to 255
Example	l_u8_wr(&Lin_Sig_Test4, 123);
Note	The name of the signal is a name defined by the user. The address defined for the signal is substituted for this. Do not call this function to write a signal which is not actually a 2- to 8-bit signal. Operation is not guaranteed when the function is called to write such data.

Signal Value Reading

void l_u16_wr(l_signal_handle sig_name, l_u16 sig)

Description	Writes a 9- to 16-bit signal
Return value	Value of the signal: 0 or 1
Argument	sig_name Name of the signal sig Value of the 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 write a signal which is not actually a 9- to 16-bit signal. Operation is not guaranteed when the function is called to write 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 data to a byte-array 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 written 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 write a signal which is not actually a byte-array signal. Operation is not guaranteed when the function is called to write such data. Also, do not set a number of bytes that extends the defined signal size. Writing does not proceed if an error occurs.

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.

Header Detection

void l_ifc_aux(l_u8 ifc_name)

Description	Detects a header
Return value	None
Argument	ifc_name Name of the interface

Example	vodi timer_isr(void)
	{
	    l_ifc_aux(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 input-capture or IRQ interrupt-driven reception. 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	Acquires a state value
Return value	State value: See section 3, “References.”
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 Frames of Raw Diagnostic Data

Reservationof the Transmission of 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, error checking is not automatically executed. However, checking should be executed on the side that calls this function.

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 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, 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, error checking is not automatically executed. However, checking should be executed on the side that calls this function.

Verification of the Transmission of a Frame of Raw Diagnostic Data

l_u8 raw_tx_status(l_u8 ifc_name)

Description	Verifies the state of the 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.

Verification of the Reception of a Frame of Raw Diagnostic Data

l_u8 ld_raw_rx_status(l_u8 ifc_name)

Description	Verifies the state of the reception FIFO buffer in terms of the reception of a frame of raw diagnostic data
Return value	The FIFO buffer holds data: LD_QUEUE_FULL An error in transfer has occurred: LD_TRANSFER_ERROR
Argument	ifc_name Name of the interface
Example	l_u8 rtn; rtn = ld_raw_rx_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 length Length of the buffer to hold the 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, 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 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 Transmission in progress: LD_IN_PROGRESS An error in transmission has occurred: 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.

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

l_u8 ld_rx_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_rx_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() ) {
         /* Some Sensor 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 Slave Driver Entry   */
const T_Lib_Slave_Handle Slave_handle = {
   Lin_Drv_Init,
   Lin_Drv_BreakIn,
   Lin_Drv_BreakInFinish,
   Lin_Drv_BitInStart,
   Lin_Drv_BitIn,
   Lin_Drv_BitInEnd,
   Lin_Drv_SyncInFinish,
   Lin_Drv_Pid_RecvReq,
   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 */
unsigned char lin_SomeCotrol_init( void )
{
   unsigned char rtn;

   rtn = 0;
   if( l_ifc_ioctl( 0, LIN_ENTRY_SLAVE_DRV, &Slave_handle ) ) {
      /* The init of the LIN slave 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 {
         /* ......Something to do */
      }
   }
   return rtn;
}
```

2.4.2 Applications

Sample codes regarding the API functions of LIN2.0 library, which are called from applications other than those for initialization and scheduling, are described in this section. The usage of data acquired by calling the API functions depends on the application, and so is not specified in this example. Contents (frames) transferred on the LIN bus are data on the states of the various nodes and data acquired by peripheral devices. Therefore, how data should be transferred and processed will depend on the configuration of the particular LIN system.

```
#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_Frm_FrameMst0_flg;
extern l_flg Lin_Frm_FrameU1_flg;
extern l_flg Lin_Frm_FrameU2_flg;
extern l_flg Lin_Frm_FrameU3_flg;
extern l_flg Lin_Frm_FrameEve0_flg;
extern l_flg Lin_Frm_FrameSlv0_flg;
extern l_flg Lin_Sig_Command_flg;
extern T_Signal Lin_Sig_Status_Slv0;
extern T_Signal Lin_Sig_Status_Slv1;
extern T_Signal Lin_Sig_Command;
void lin_application( void )
{
   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;

   /* Has a frame been received by the slave node? */
   if( 0 != l_flg_tst(&Lin_Frm_FrameU1_flg) ) {
      l_flg_clr( &Lin_Frm_FrameU1_flg );
      /* Something is done */
   }
   else if( 0 != l_flg_tst(&Lin_Frm_FrameMst0_flg) ) {
      l_flg_clr( &Lin_Frm_FrameMst0_flg );
      /* Something is done */
   }

   /* Has a frame been received by the slave node? */
   if( 0 != l_flg_tst(&Lin_Frm_FrameU1_flg) ) {
      l_flg_clr( &Lin_Frm_FrameU1_flg );
      /* Something is done */
   }
   else if( 0 != l_flg_tst(&Lin_Frm_FrameMst0_flg) ) {
      l_flg_clr( &Lin_Frm_FrameMst0_flg );
      /* Something is done */
   }
```

```
      /* Has a frame been transmitted by the slave node? */
     if( 0 != l_flg_tst(&Lin_Frm_FrameU2_flg) ) {
     l_flg_clr( &Lin_Frm_FrameU2_flg );
        /* Something is done */
     }
     /* Has a frame been transmitted by the slave node? */
     else if( 0 != l_flg_tst(&Lin_Frm_FrameU3_flg) ) {
        l_flg_clr( &Lin_Frm_FrameU3_flg );
        /* Something is done */
     }
     else if( 0 != l_flg_tst(&Lin_Frm_FrameEve0_flg) ) {
        l_flg_clr( &Lin_Frm_FrameEve0_flg );
        /* Something is done */
     }
     else if( 0 != l_flg_tst(&Lin_Frm_FrameSlv0_flg) ) {
        l_flg_clr( &Lin_Frm_FrameSlv0_flg );
        /* Something is done */
     }

     status.Word = l_ifc_read_status( 0 );

     if( status.Bit.errorrsp ) {
        /* Processing in response to the error */
     }

     if( LD_DATA_AVAILABLE == ld_raw_rx_status(0) ) {
        ld_get_raw( 0, data );
     }

     /* Has a signal been sent by the master node? */
     if( 0 != l_flg_tst(&Lin_Sig_Command_flg) ) {
        l_flg_clr( &Lin_Sig_Command_flg );
        if( 0x1234u == l_u16_rd(&Lin_Sig_Command) ) {
           l_u16_wr( &Lin_Sig_Status_Slv0, 0x0101u );
           l_u16_wr( &Lin_Sig_Status_Slv1, 0x0201u );
              /* Something is done    */
        }
        else if( 0x5678u == l_u16_rd(&Lin_Sig_Command) ) {
           l_u16_wr( &Lin_Sig_Status_Slv0, 0x0100u );
           l_u16_wr( &Lin_Sig_Status_Slv1, 0x0200u );
              /* Something is done */
         }
     }

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

     return;
  }
```

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
- H8/36024 Group H8/36014 Group Hardware Manual: REJ09B0025-0400

Revision Record

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