Introduction

This application note describes how to use LIN and UART controllers of various Rensas microcontroller products.

Several different LIN and UART controller types are available; so this application note is collecting frequently asked questions and hot topics for all of them. By using the index, the user may locate the answers in one or several chapters. Nevertheless, the content of this application note does not make any claim to be complete.

Due to this, the application note may be updated without further notice in shorter time intevals.
Proposals for improvement are always highly welcome.

Target Device

earlier series:	U(A)RT*x*	UART types *A-D*
V850/FK3:	MLM	LIN Controller types
V850/Xx4:	LMA/U(A)RTE	LIN Controller types
RL78/X1x:	RLIN3/U(A)RTF	LIN Controller types
RH850/X1x:	RLIN3	LIN Controller types

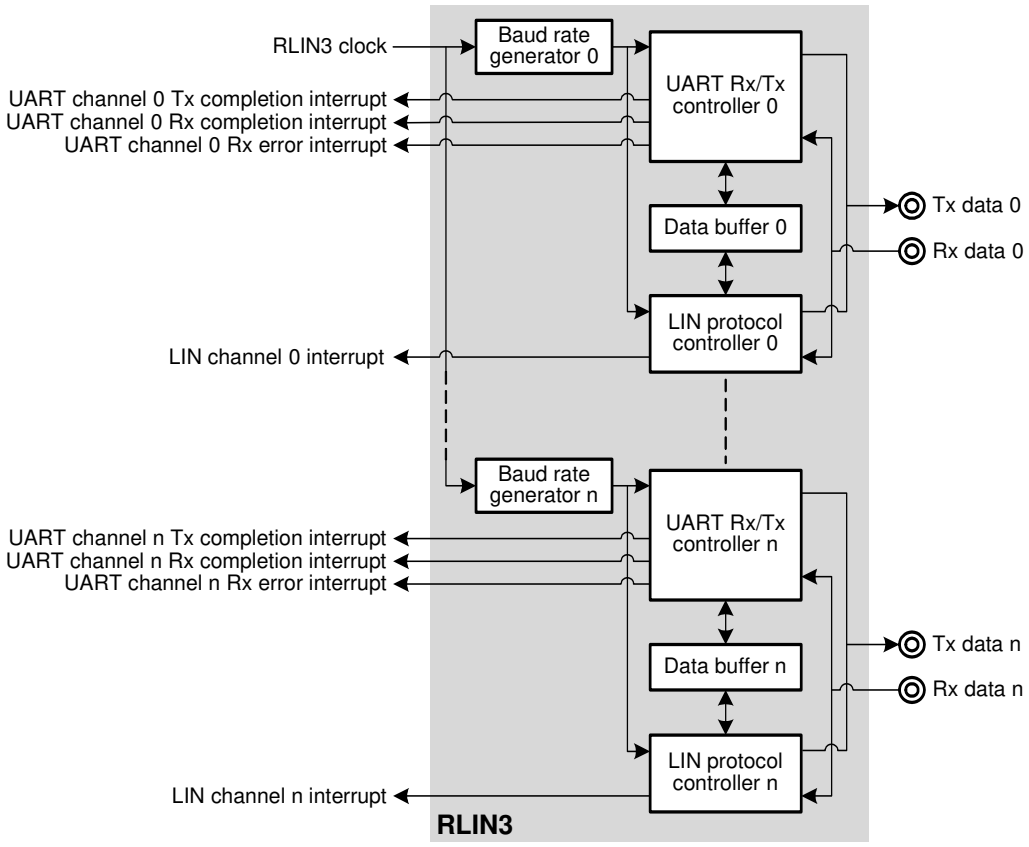

Figure 1.1 State of the art LIN Controller RLIN3

Note: Subsequent pages may be partly blank or have interleaved chapter numbering.
This is by intention, as this application note is continuously improved.

Table of Contents

Terminology Index

Issue Solving Proposal Index

2. LIN Applications

2.1 Using LMA, U(A)RTE or U(A)RTF as LIN Master and U(A)RT*x* as LIN Slave

The following applies to any NEC/Renesas UART types U(A)RT*A* to U(A)RT*F*, when used in conjunction with LMA, U(A)RTE and U(A)RTF.

Note: Nomenclature of function names has changed from formerly "UART" to "URT" in the applicable manuals.

The UART reports the completion of a reception, as soon as the last data bit of the current frame has been sampled. As the sampling point is located in the middle of the bit, the reception interrupt occurs just a few peripheral clocks later.
On the other hand, the bit consistency is checked (if enabled) by the LIN master at about 70% to 80% of the bit time. In order to be within the LIN specification, the bit consistency checking functionality of the LIN master must be enabled.
Consequently, if a LIN master based on LMA, U(A)RTE, U(A)RTF is combined with any U(A)RT*x* slave, the slave will report the reception at a time, where the master has not yet finished transmission of the same LIN byte, here, the PID.

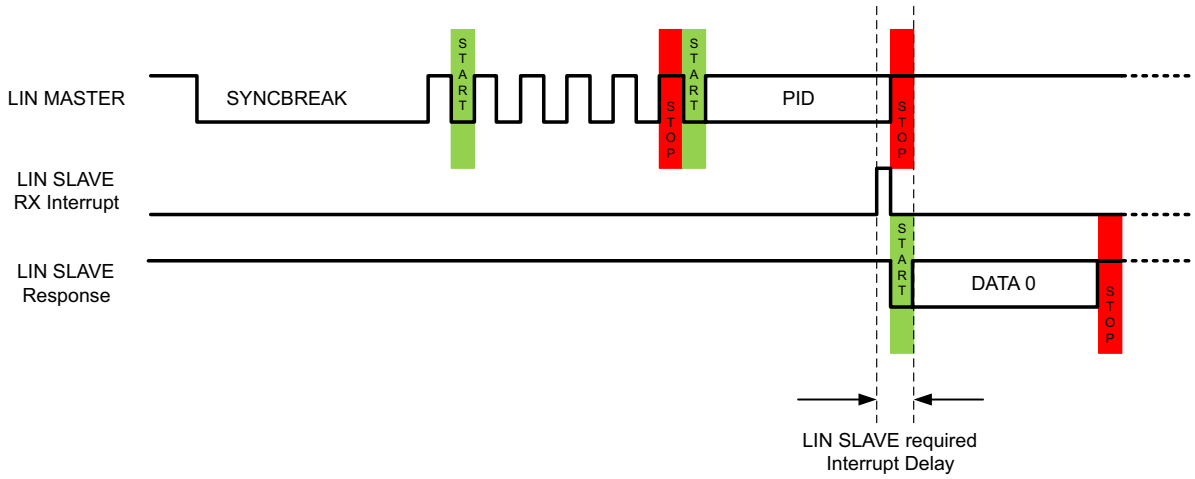

Figure 2.1 LIN Master / Slave Coincidence

Therefore, the U(A)RT*x* slave must have implemented a delay function after LIN reception, which suppresses the transmission of data right after a reception. Otherwise, this transmission would coincide with the ongoing reception from the master; and the LIN master would detect this as a bit error.

Implementations of LIN slaves using RLIN3 are not affected by this issue, because RLIN3 considers this situation in its state engine.

2.2 Distinguishing between Framing Error and valid BREAK Condition

The following situation requires special attention for a *LIN slave* implementation, independent of any LIN controller hardware. It is addressing the “Response Error” reporting of a LIN slave, when having to distinguish between a frame error at the STOP bit of the first data field of a response or a new BREAK detection.

It is a feature of LIN, that the response of a frame can be omitted by just starting another frame, doing so by sending a new BREAK field. A LIN slave, which is waiting to receive the response of a frame in this case must stop its response waiting, and recognize the new frame.

The recognition of a BREAK field when waiting for a response usually causes a framing error in the LIN slave controller. At this point, the LIN slave has to take an action on this.
If already at least one byte of the response was received, this framing error can be easily categorized to be a real framing error, because the abortion of a response by a BREAK field is not allowed. To detect this, LIN slave controllers have the indication flag of the first response data byte in their register set.
If no response byte has been received yet, at the point in time when the BREAK is sent, then the LIN slave has two options to categorize the framing error:

(1) A BREAK has occurred, and the framing error must not be reported as a Response Error.
(2) In the first data byte, which has the value of 0x00, a frame error is detected, means, the STOP bit is inverted. Thus, a Response Error must be reported.

At the point of time, when the frame error occurs, the LIN slave cannot distinguish between these two cases.
A distinguishment is possible at latest, after the new header (BREAK+SYNC+ID) has been received completely.
So, if the LIN slave shall be able to distinguish the two cases, special delay circuitry of framing errors would be required in hardware, or an additional time condition must be checked in software by using a hardware timer.

In the figure below, the situation is shown with a BREAK length of 10 bits, which is the minimum to be detectable by LIN slaves.

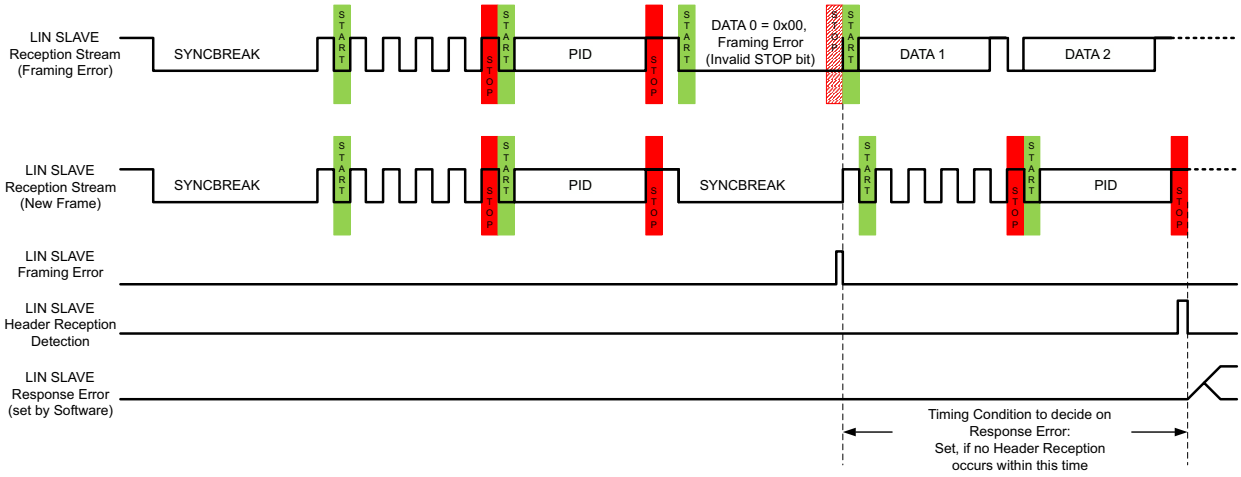

Figure 2.2 Two cases of reception to decide on “Response Error” flagging

Currently, the detection and distinguishment of these two cases is not available in the LIN controller hardware. Therefore, if required, the timing condition shown in the figure needs to be evaluated by software; it is recommended to use a hardware timer instance for this purpose.
At the framing error interrupt, the timer shall be started and set to a timeout condition, which is longer than SYNC and PID of the LIN header. If no LIN slave header reception interrupt occurs, before the timer indicates a timeout, then the last frame has had a framing error and the “Response Error” flag can be indicated to upper software layers and in the LIN response data stream to the LIN master.

2.3 LIN Synchronization

LIN Synchronization takes place in a LIN Slave, which synchronizes its bit rate to the LIN Master. As LMA and RLIN2 have no LIN Slave functionality, they are not considered in this chapter.

2.3.1 Legacy Serial Interfaces UARTA to UARTE

These interfaces are not supporting automatic or implicit LIN bus synchronization.

The UART needs to be connected to an additional internal timer unit with edge detection input from the RXD port. By software implementation, the measurement of the SYNC field of the LIN frame must be performed with the support of the internal timer unit. In detail, the steps to be done are the following:

(1) Enable the detection and reception of the BREAK field of the LIN frame.
Set the “SBF reception trigger” to receive an event, after the BREAK field has been detected.

(2) After the BREAK field has been detected successfully, measure at least two subsequent bits of the following SYNC field using the timer in interval measurement mode. The start bit and the LSB/LSB+1 bits of the SYNC field are forming a 0-1-0 pulse, where the width of either part can be measured using the hardware timer.
Concurrently, start the UART reception with the predefined bit rate, which should already be accurate enough for an unadjusted reception of the SYNC field.

(3) After the timer interrupt for having completed the measurement, calculate the bit rate of the frame by software. Intermediately, stop the UART, set the bit rate pre-scaler register of the UART according to the result and restart it again. This step must happen very quickly without any interruption, otherwise the next field of the LIN frame might be missed.

(4) The next fields of the LIN bus then should be received with the adjusted bit rate, which also shall be the initial setting for the next following frame.

Limitations of the synchronization method:

- The bit rate of the received frame must already be known roughly with a maximum tolerance of 5%; otherwise the reception of the SYNC field might fail and the error condition of the UART may cause that subsequent bytes cannot be received despite of a bit rate measurement.
Reason: the UART sampling point is at 50% of the bit, and 10 bits of a LIN byte are received after only one initial synchronization using the falling edge of the start bit.

- If the bits of the SYNC field of the LIN frame do not have equal width, even though the SYNC field has its exact length, then the synchronization may fail.
Reason: the method only uses two bits of the SYNC field to get the bit rate; if these are malformed, then the measurement will have a wrong result.

- If the bit rate changes or varies during the LIN frame after the SYNC field, this cannot be recognized.
Reason: the measurement only takes place at the SYNC field, and during the remaining frame, the tolerance of the bit rate must not exceed the given LIN bus limits, because no further synchronization during the LIN frame takes place.

2.3.2 Serial Interfaces UARTF and RLIN3

UARTF and RLIN3 are supporting an internal automatic synchronization mechanism, which can be activated in LIN Slave mode optionally. The synchronization method of this mechanism is described within this chapter.

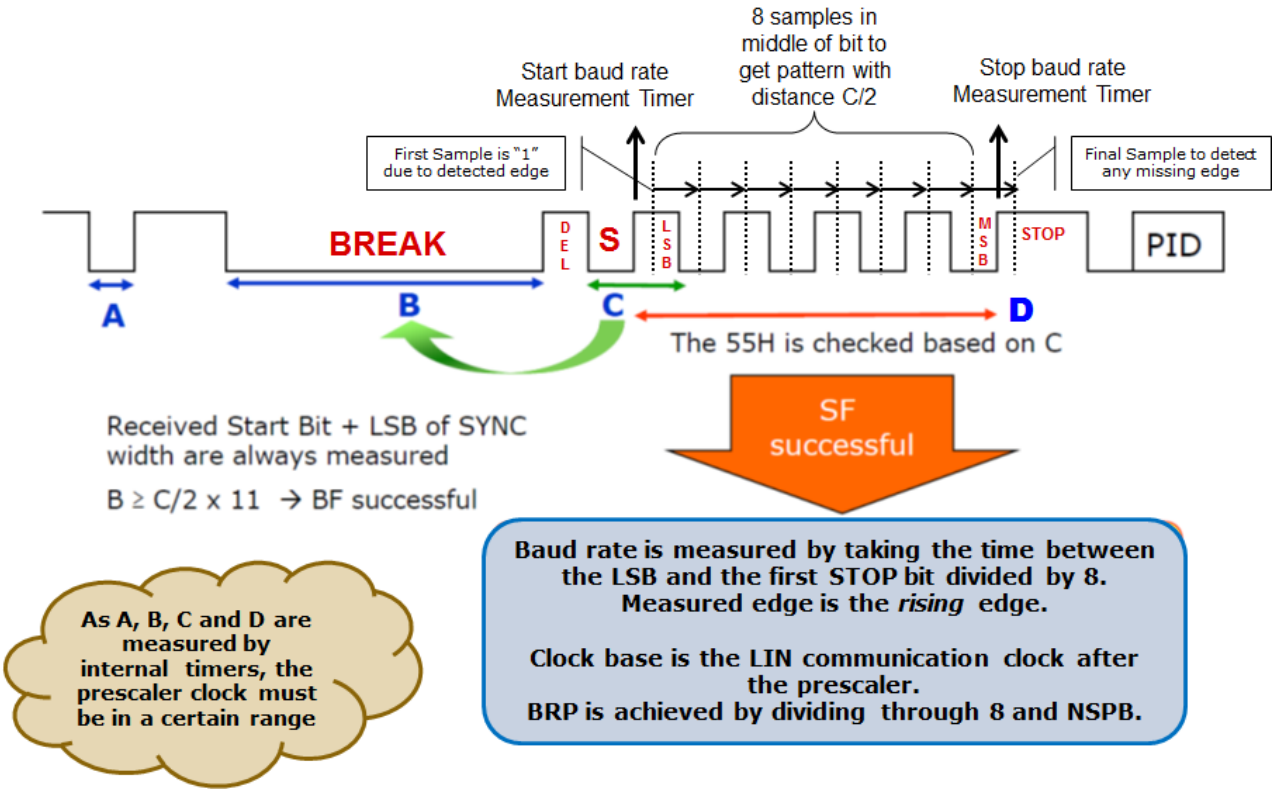

Figure 2.3 UARTF and RLIN3 LIN Slave Automatic Frame Synchronization

The synchronization has several steps (A to D), and performs several internal calculations and verifications internally. Initial settings with rough bit rate pre-set are required, otherwise some verifications and calculations may fail in principle. Therefore, even though an automatic bit rate synchronization is provided, ***this does not allow*** to work without any rough initialization values for the bit rate.

(A) The width of the BREAK field is checked for its minimum, to be recognized as valid.
(B) After detecting a minimum width of the BREAK field, the LIN controller gets ready to receive the BREAK delimiter and the SYNC field.
(C) The distance between the first two falling edges within the SYNC field (start bit and LSB) is measured and compared with the length of the BREAK field. The frame is accepted to be a LIN frame so far, if the condition “B >= C/2 * 11” is fulfilled.
This measurement is done using an internal timer. To avoid overflow of this timer, the minimum bit rate and maximum clock speed of the LIN controller have to be in a certain range.
The length of C/2 is used to verify the bit value pattern of the SYNC field to be 0x55.
At the same time, another timer is started to measure the SYNC field width.
(D) After eight edges of the SYNC field, the measurement timer for the SYNC field length (D) is stopped.

(E) Final steps to verify and set the measured bit rate:
The sampled bit value pattern of the SYNC field must be 0x55.
The measured interval D is divided by eight and divided by the amount of samples per bit (NSPB) of the LIN bit composition. NSPB is a customizable setting by the user and defines the bit and sampling point position granularity (sub-segmentation of a bit).

Limitations of the synchronization method:

- The bit rate of the received frame must already be known roughly; otherwise the detection of the BREAK field might fail.
  Reason: The minimum width detection of the BREAK field is a preconditon to detect the LIN frame. Typically, a tolerance of 20% can be acceptable here.

- If the first two bits of the SYNC field of the LIN frame do not have equal width, even though the SYNC field has its exact length, then the synchronization may fail.
  Reason: the method uses two bits of the SYNC field to get the rough bit rate to verify the SYNC field bit value pattern; if these are malformed, then the SYNC field bit value pattern verification may fail.

- If the bit rate changes or varies during the LIN frame after the SYNC field, this cannot be recognized.
  Reason: the measurement only takes place at the SYNC field, and during the remaining frame, the tolerance of the bit rate must not exceed the given LIN bus limits, because no further synchronization during the LIN frame takes place.

2.4 Precision of the LIN / UART Bit Rate

The precision of a set bit rate for the LIN / UART controllers depends on several factors, which are defined in the following:

(1) Communication clock frequency (operation clock of the LIN / UART data link layer), f_c
(2) Number of bits of the SYNC field used to count for the bit rate, N_{SYNC}
(3) Number of segments (quanta) within a bit to set the resolution of the sampling point, N_{SPB}
(4) Amount of division factor by the bit rate pre-scaler, N_{BRP}

2.4.1 Legacy Serial Interfaces UARTA to UARTE

Within the legacy serial interfaces UARTA to UARTE, there are no dedicated functions implemented for bit rate measurement. If the bit rate shall be measured, then the method is chosen by the user's additional effort, and thus it cannot be considered within this document.

However, when setting the bit rate, this happens in steps of the bit rate pre-scaler. Therefore, the following properties concerning accuracy ***A*** will apply, and the deviation of the user method to get the bit rate by measurement must be added:

$$\mathbf{A} = \frac{1}{N_{BRP}}$$

2.4.2 Serial Interfaces UARTF and RLIN3

When measuring the bit rate by hardware, several components of the accuracy exist, which must be added to evaluate the total accuracy ***A***:

$$\mathbf{A} = A_S$$

with A_M: Measurement accuracy
A_C: Calculation accuracy, depends on A_M
A_S: Setting accuracy, depends on A_C

All three components of the total accuracy are listed separately, because often asked in customer queries.
They are calculated as follows:

$$A_M = \frac{\text{bitrate}}{N_{SYNC} \times f_c}$$

Rationale: The amount of communication clock cycles which fit into the measurement period is set into relation with the bit rate.

$$A_C = A_M \times N_{SYNC}$$

Rationale: When calculating the time of a single bit, the timer result is truncated by lower bits.

$$A_S = A_C \times N_{SPB}$$

Rationale: When calculating the time of a bit quantum, the timer result is again truncated by lower bits.

3. Sample Software Description

3.1 Abstract

This chapter is describing the software packages, which are available upon request as usage examples (so-called “Sample Software”) for the currently available LIN / UART controllers on microcontroller devices.

The software and the description in this chapter are given free of charge, and therefore some conditions apply:
• No guarantee of function for customer specific hardware implementations
• No liability from Renesas side on consequent issues (such as failures, loss of data, injury), caused by its usage
• No allowance to use the software in any commercial product
• No copying of the software without mentioning these conditions
• No announcement of any changes from Renesas side
• No tracking of issues or versioning
• No upgrade compatibility when replaced

The following is covered by this chapter:
• Lower LIN / UART driver functionality
• Mapping of the lower LIN / UART driver to specific device properties
• Applications based on lower LIN / UART driver functions, including a serial or debugger based monitor program

As a general approach, this documentation is not describing each and every step and definition of the software. Instead, we are describing the functionality by showing the algorithmic content, hereby highlighting certain topics which are of certain interest.
All other information the user shall determine from the software programs directly.

The documentation is made for several kinds of LIN / UART controllers, as it is a target to have the API as much close and similar, we are distinguishing among the LIN / UART controllers in sub-chapters. Each LIN / UART controller has a certain prefix, this allows that several software parts for different controller types can be integrated at the same time. The prefix is indicated by <xxx> in this description.
Not all functions are available for all LIN / UART controller types (in case not mentioned, functions are not implemented).

3.2 Supported LIN / UART Controller Hardware

In the current state of this documentation release, the following LIN / UART controller types on device series can be supported:

Table 3.1 Sample software availability on request for LIN / UART Controller Hardware

Device Class	Device Family Members	LIN / UART Controller Type	Prefix <xxx>[a]	Index[b]
V850	FG2, FJ2, CARGATE+	UARTA ([c])	UARTA	(A)
	FJ3, FK3	UARTD ([d])	UARTD	(D)
	FG4-L, DN4	UARTE ([e])	UARTE	(E)
78K0R	FJ3	UARTF	UARTF	(F)
RL78	D1A, F12			
	F13, F14, F15	RLIN3	RLIN3	(R3)
RH850	F1A, F1L, F1M, P1M-E, P1H-C			
	F1H, F1K	RLIN2 ([f])	RLIN2	(R2)

a. The prefix is used in function names and *#define* constant names of the different drivers of the CAN controller types.
b. Label referenced by function chapters (“implementations”), to show if available/implemented.
c. Sample Software for UARTA is no longer maintained.
Existing packages can be provided “as is”, but without support and guarantee of operation.
d. Sample Software for UARTD is no longer maintained.
Existing packages can be provided “as is”, but without support and guarantee of operation.
e. Sample Software for UARTE is no longer maintained.
Existing packages can be provided “as is”, but without support and guarantee of operation.
f. RLIN2 does not support UART functionality (LIN Master only).

3.3 Lower Level LIN / UART Driver Functionality

3.3.1 Overview

Purpose of a lower level LIN / UART driver is to implement the hardware based abstraction into a certain function set, which we will further call “API” (Application Programming Interface).
The main tasks in a LIN / UART driver for this API are:
• Initialization of the LIN / UART controller environment, such as port and clock settings
• Initialization and configuration of the LIN / UART controller
• Reset of the LIN / UART controller
• Providing means of sending and receiving LIN messages, LIN frame parts or UART bytes
• Providing means of interrupt processing, including hooks for interrupt handling expansions
• Providing status information of the LIN / UART controller
• Reading and clearing error states of the LIN / UART controller

The lower level LIN / UART driver resides in the files *_p.c* with API **_p.h* and definition in **.h*.

3.3.2 Environmental Initialization

Within this driver description, we assume that the clock speed of the LIN / UART controller is fixed and given by some setting of the clock system of a specific device. Therefore, we will have a certain definition constant, which specifies the clock speed, and this constant will have an influence on the bit rate setting, for example.
Other timing we are doing by simple loops with a maximum loop count in order to have a safe way out of hardware failures.
Summarized, for clock settings, the lower LIN / UART driver has no API. The clock speed is a known constant and timeout loops have to be adjusted by specifying the maximum amount of loops (which in fact has a dependency on the used clock speed, of course).
Regarding interrupts, the lower level driver typically contains 3 interrupt routines, which also support callback mechanisms. By mapping (see 3.4), the interrupt sources are bound with the interrupt controller.
Each LIN / UART channel has two ports to the outside world, which is the *transceiver*. A transceiver is doing the physical layer adaptation from the external UART or LIN bus signal into the *transmit* and *receive* direction ports of the LIN / UART controller. The LIN controller requires that its transmission get visible on the LIN bus, and the LIN bus gets visible (read back) on its reception path. This is provided by the *LIN transceiver*.
In order to connect the external transceiver, the two I/O ports need to become initialized, before starting the LIN / UART controller.

3.3.3 Used Types

The following data types are used in the API, which are bound to elementary ANSI C data types. Parameters and functions are named with corresponding endings in order to indicate the data type.

Table 3.2 Used Types in API

Data Type	Ending	ANSI Type	Description
u08	_u08	unsigned char	8 bits
s08	_s08	signed char	7 bits plus sign bit at MSB position
u16	_u16	unsigned short	16 bits
s16	_s16	signed short	15 bits plus sign bit at MSB position
u32	_u32	unsigned long	32 bits
s32	_s32	signed long	31 bits plus sign bit at MSB position
flt	_flt	float	floating point decimal
dbl	_dbl	double	floating point double precision decimal
pu08	_pu08	unsigned char *	pointer to u08 (pass by reference)
pu16	_pu16	unsigned short *	pointer to u16 (pass by reference)
pu32	_pu32	unsigned long *	pointer to u32 (pass by reference)
bit	_bit	enum {false, true}	boolean enumeration with false = 0, true = 1
[pointer to structure]	-	-	see documentation of the function
void	-	void	function returns no value

3.3.4 Port I/O Initialization

3.3.4.1 <xxx>_PortEnable( )

Implementations: (A), (D), (E), (F), (R2), (R3).

(1) Implementations (A), (D), (E), (F), (R3)

(1-1) Parameters

UnitNumber_u08: Selected LIN / UART Controller

(1-2) Return Values

<*xxx*>*_ERROR* on parameter failures, otherwise <*xxx*>*_OK*.

(1-3) Functional Description

The function enables (activates) the port I/O for the dedicated LIN / UART controller unit.
A standard port support library is called to perform the function.
For each used LIN / UART controller unit, the following dedicated *#define* constants must be set in the driver's mapping definition (see 3.4 for details and file information):

#define <xxx>_PORT_RXD<*unit*>	PORT*_p*
#define <xxx>_PORT_BIT_RXD<*unit*>	BIT*_r*
#define <xxx>_PORT_FUNC_RXD<*unit*>	PORT_FUNCTION_ALTLV*_a*
#define <xxx>_PORT_TXD<*unit*>	PORT*_q*
#define <xxx>_PORT_BIT_TXD<*unit*>	BIT*_t*
#define <xxx>_PORT_FUNC_TXD<*unit*>	PORT_FUNCTION_ALTLV*_b*

These settings will bind the port I/O lines *p.r* and *q.t* in the alternate port assignment level *a* vs. *b* to the LIN / UART controller unit <*unit*>; where *p, q, r, t, a* and *b* are decimal text characters (for numeric ports and port bits of the specific device).

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08: Selected LIN Controller
ChannelNumber_u08: Selected LIN Controller channel

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function enables (activates) the port I/O for the dedicated LIN controller unit and channel.
A standard port support library is called to perform the function.
For each used LIN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 3.4 for details and file information):

#define <xxx>_PORT_M<*unit*>RX<*channel*>	PORT_*p*
#define <xxx>_PORT_BIT_M<*unit*>RX<*channel*>	BIT_*r*
#define <xxx>_PORT_FUNC_M<*unit*>RX<*channel*>	PORT_FUNCTION_ALTLV_*a*
#define <xxx>_PORT_M<*unit*>TX<*channel*>	PORT_*q*
#define <xxx>_PORT_BIT_M<*unit*>TX<*channel*>	BIT_*t*
#define <xxx>_PORT_FUNC_M<*unit*>TX<*channel*>	PORT_FUNCTION_ALTLV_*b*

These settings will bind the port I/O lines *p.r* and *q.t* in the alternate port assignment level *a* vs. *b* to the LIN controller unit <*unit*>, channel <*channel*>; where *p, q, r, t, a* and *b* are decimal text characters (for numeric ports and port bits of the specific device).

3.3.4.2 <xxx>_PortDisable( )

Implementations: (A), (B), (D), (E), (R2), (R3).

(1) Implementations (A), (B), (D), (E), (R3)

(1-1) Parameters

UnitNumber_u08: Selected LIN / UART Controller

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

The function disables (deactivates) the port I/O for the dedicated LIN / UART controller unit.
A standard port support library is called to perform the function.
The *#define* constants used for this function are the same as for function <xxx>_PortEnable( ).

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08: Selected LIN Controller
ChannelNumber_u08: Selected LIN Controller channel

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function disables (deactivates) the port I/O for the dedicated LIN controller unit and channel.
A standard port support library is called to perform the function.
The *#define* constants used for this function are the same as for function <xxx>_PortEnable( ).

3.3.5 LIN / UART Controller Initialization and Configuration

3.3.5.1 <xxx>_Configuration( ), <xxx>_LegacyConfig( )

Implementations: (A), (D), (E), (F), (R3).

(1) Implementations (A), (D), (E), (F), (R3)

The function name depends on the implementation:

(A), (D), (E), (F):	<xxx>_Configuration( )
(R3):	<xxx>_LegacyConfig( )

(1-1) Parameters

UnitNumber_u08:	Selected LIN / UART controller
Oscillator_Frequency_flt:	Frequency of the LIN / UART controller communication clock in Hz.
Baudrate_u32:	Baud rate of UART protocol in bit/s, as an integer value.
Parity_u08:	One of <xxx>_PARITY_NONE, <xxx>_PARITY_ZERO, <xxx>_PARITY_ODD or <xxx>_PARITY_EVEN
CharLen_u08:	7, 8 or 9-bit length of one UART data element: One of <xxx>_CHARLEN_7BITS, <xxx>_CHARLEN_8BITS or <xxx>_CHARLEN_9BITS
StopBits_u08:	One or two stop bits of the UART protocol: One of <xxx>_ONESTOPBIT or <xxx>_TWOSTOPBITS
Direction_u08:	Bit ordering of the UART protocol: One of <xxx>_DIR_MSBFIRST or <xxx>_DIR_LSBFIRST

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function name for (R3) implementation is different, because for (R3), additional parameters are available which are covered by the function <xxx>_UARTModeConfig( ).
The function initializes and starts the UART mode of the LIN / UART controller and sets the given communication parameter properties beforehand.
The appropriate pre-scaler and bit rate generator settings are calculated automatically by the function, so that best precision is achieved.

3.3.5.2 <xxx>_UARTModeConfig( )

Implementations: (R3).

(1) Implementations (R3)

(1-1) Parameters

UnitNumber_u08:	Selected LIN / UART Controller
Oscillator_Frequency_flt:	Frequency of the LIN / UART controller communication clock in Hz.
Baudrate_u32:	Baud rate of UART protocol in bit/s, as an integer value.
SamplesPerBit_u08:	Number of time quanta per bit (resolution of sample point): Allowed values are 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16.
Parity_u08:	One of <xxx>_PARITY_NONE, <xxx>_PARITY_ZERO, <xxx>_PARITY_ODD or <xxx>_PARITY_EVEN
CharLen_u08:	7, 8 or 9-bit length of one UART data element: One of <xxx>_CHARLEN_7BITS, <xxx>_CHARLEN_8BITS or <xxx>_CHARLEN_9BITS
StopBits_u08:	One or two stop bits of the UART protocol: One of <xxx>_ONESTOPBIT or <xxx>_TWOSTOPBITS
Direction_u08:	Bit ordering of the UART protocol: One of <xxx>_DIR_MSBFIRST or <xxx>_DIR_LSBFIRST
FilterMode_u08:	<xxx>_SAMPLE_SINGLE: one sample per bit <xxx>_SAMPLE_TRIPLE: three samples per bit around sample point
InterruptMode_u08:	<xxx>_SINGLE_INT: all interrupts are indicated by one interrupt <xxx>_TRIPLE_INT: separated interrupts for TX, RX and error

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function initializes and starts the UART mode of the LIN / UART controller and sets the given communication parameter properties beforehand.
The appropriate pre-scaler and bit rate generator settings are calculated automatically by the function, so that best precision is achieved.
In addition to the function <xxx>_LegacyConfig(), specific additional parameters for (R3) type controllers are available.

3.3.5.3 <xxx>_LINSetConfig( )

Implementations: (A), (D), (E), (F), (R2), (R3).

(1) Implementations (A), (D), (E)

(1-1) Parameters

UnitNumber_u08:	Selected LIN / UART Controller
DataDirection_u08:	Bit ordering of the LIN protocol: One of <xxx>_DIR_MSBFIRST or <xxx>_DIR_LSBFIRST
SignalLevel_u08:	Valid for both RX and TX signals: <xxx>_TXDNORMAL, <xxx>_RXDNORMAL: Normal level <xxx>_TXDINVERT, <xxx>_RXDINVERT: Inverted level
SyncBreakLength_u08:	Number of bits for *Break* signal generation: <xxx>_SBF13BITS ... <xxx>_SBF20BITS

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

The function initializes and starts the LIN mode of the LIN / UART controller and sets the given communication parameter properties beforehand.
This function has to be called after <xxx>_Configuration(), in order to set additional LIN mode properties and enable the LIN operation mode.
The settings of SignalLevel_u08 and DataDirection_u08 are beyond the LIN standard and can be used for additional testing or specific transceiver hardware adaptation.

(2) Implementations (F)

(2-1) Parameters

UnitNumber_u08:	Selected LIN / UART Controller
DataDirection_u08:	Bit ordering of the LIN protocol: One of <xxx>_DIR_MSBFIRST or <xxx>_DIR_LSBFIRST
SignalLevel_u08:	Valid for both RX and TX signals: <xxx>_TXDNORMAL, <xxx>_RXDNORMAL: Normal level <xxx>_TXDINVERT, <xxx>_RXDINVERT: Inverted level
SyncBreakLength_u08:	Number of bits for *Break* signal generation: <xxx>_SBF13BITS ... <xxx>_SBF20BITS
OperationMode_u08:	<xxx>_MODE_LINMASTER: LIN Master Mode operation <xxx>_MODE_LINSLAVE: LIN Slave Mode operation <xxx>_MODE_NORMAL: UART Mode operation
ExpansionMode_u08:	Activates 9-bit expansion mode for UART Mode operation, if set.
ExpansionBitValue_u08:	Sets level of expansion bit (9th bit) for detection.
ExpansionComparison_u08:	Activates expansion bit (9th bit) checking for addressing.
IDParityCheck_u08:	Activates LIN ID parity check in LIN modes reception, if set.
AutoChecksum_u08:	Activates automatic checksum calculation in LIN modes, if set.
DataConsistencyCheck_u08:	Verifies TX value by reading back RX from the transceiver, if set.

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function initializes and starts the LIN or UART modes of the LIN / UART controller and sets the given additional communication parameter properties beforehand.
This function has to be called after <xxx>_Configuration(), in order to set additional LIN or UART mode properties and enable the LIN / UART operation mode.
The settings of SignalLevel_u08 and DataDirection_u08 are beyond the LIN standard and can be used for additional testing or specific transceiver hardware adaptation.
The settings of ExpansionMode_u08, ExpansionBitValue_u08 and ExpansionComparison_u08 are valid for the 9-bit UART mode only, and must not be activated in LIN operation modes.

(3) Implementations (R2)

(3-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel
Oscillator_Frequency_flt:	Frequency of the LIN controller communication clock in Hz.
LINStandard_u08:	<xxx>_LINSTANDARD_V1: Operation for LIN V1.x standard. <xxx>_LINSTANDARD_V2: Operation for LIN V2.x standard.
Baudrate_u32:	Baud rate of LIN protocol in bit/s, as an integer value.
ClockSelection_u08:	Selection of pre-scaled clock of the LIN controller: <xxx>_CLOCK_FA: communication clock divided by 1. <xxx>_CLOCK_FB: communication clock divided by 2. <xxx>_CLOCK_FC: communication clock divided by 8. <xxx>_CLOCK_FD: communication clock divided by 2, but using a separate bit rate generator.
BreakLength_u08:	Number of bits for *Break* signal generation: <xxx>_SBF13BITS ... <xxx>_SBF28BITS
BreakDelimiter_u08:	Length of break field delimiter in bits: <xxx>_BDT1BITS ... <xxx>_BDT4BITS
HeaderSpace_u08:	Space between LIN header and LIN response in bits: <xxx>_SPACE_HDRRESP_0BITS ... <xxx>_SPACE_HDRRESP_7BITS
InterByteSpace_u08:	Space between data and checksum bytes of LIN response in bits: <xxx>_SPACE_RESBYTE_0BITS ... <xxx>_SPACE_RESBYTE_3BITS

(3-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(3-3) Functional Description

The function initializes and starts the LIN Master mode of the LIN controller and sets the given communication parameter properties beforehand. Clock selection settings are common for all channels. Using <xxx>_CLOCK_FD allows additional bit rates when using several LIN Master channels. To avoid side effects when calling the function for several channels, check proper combinations of clock selection and bit rate beforehand by consulting the manual. Otherwise, a bit rate setting of a channel may invalidate the setting of another channel.

(4) Implementations (R3)

(4-1) Parameters

UnitNumber_u08:	Selected LIN Controller
Oscillator_Frequency_flt:	Frequency of the LIN controller communication clock in Hz.
OperationMode_u08:	<xxx>_MODE_LINMASTER: LIN Master operation <xxx>_MODE_LINSLAVE_FIX: LIN Slave fixed bit rate operation <xxx>_MODE_LINSLAVE_AUTO: LIN Slave auto bit rate operation (resynchronizes deviations)
LINStandard_u08:	<xxx>_LINSTANDARD_V1: Operation for LIN V1.x standard. <xxx>_LINSTANDARD_V2: Operation for LIN V2.x standard.
Baudrate_u32:	Baud rate of LIN protocol in bit/s, as an integer value.
SamplesPerBit_u08:	Number of time quanta per bit (resolution of sample point).
ClockSelection_u08:	Selection of pre-scaled clock of the LIN controller: <xxx>_CLOCK_FA: communication clock divided by 1. <xxx>_CLOCK_FB: communication clock divided by 2. <xxx>_CLOCK_FC: communication clock divided by 8. <xxx>_CLOCK_FD: communication clock divided by 2, but using a separate bit rate generator, when using LIN Master mode. In LIN Slave fix bit-rate mode, only <xxx>_CLOCK_FA is allowed.
BreakDetectionLength_u08:	Length of break signal detection in LIN Slave modes: <xxx>_BFSLAVE_SHORT: 10 bits (auto), 9.5 bits (fixed bit rate) <xxx>_BFSLAVE_LONG: 11 bits (auto), 10.5 bits (fixed bit rate)
BreakLength_u08:	Number of bits for *Break* signal generation: <xxx>_SBF13BITS ... <xxx>_SBF28BITS
BreakDelimiter_u08:	Length of break field delimiter in bits: <xxx>_BDT1BITS ... <xxx>_BDT4BITS
HeaderSpace_u08:	Space between LIN header and LIN response in bits: <xxx>_SPACE_HDRRESP_0BITS ... <xxx>_SPACE_HDRRESP_7BITS
InterByteSpace_u08:	Space between data and checksum bytes of LIN response in bits: <xxx>_SPACE_RESBYTE_0BITS ... <xxx>_SPACE_RESBYTE_3BITS
FilterMode_u08:	<xxx>_SAMPLE_SINGLE: one sample per bit <xxx>_SAMPLE_TRIPLE: three samples per bit around sample point
InterruptMode_u08:	<xxx>_SINGLE_INT: all interrupts are indicated by one interrupt <xxx>_TRIPLE_INT: separated interrupts for TX, RX and error

(4-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(4-3) Functional Description

The function initializes and starts the LIN Master and Slave modes of the LIN controller and sets the given communication parameter properties beforehand.
Samples per bit: not all values are allowed in each mode. Consult the user's manual for details.

3.3.6 LIN / UART Controller Interrupt Management

3.3.6.1 <xxx>_CreateInterrupt( )

Implementations: (A), (D), (E), (F), (R2), (R3).

(1) Implementations (A), (D), (E), (R3)

(1-1) Parameters

UnitNumber_u08:	Selected UART / LIN Controller
InterruptSource_u08:	Selected interrupt source
InterruptLevel_u16:	Interrupt enable/level for interrupt controller setting
FunctionVector:	Pointer to a function returning void as a callback function

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

For each unit of a UART / LIN controller, each interrupt source can be selected by InterruptSource_u08 and their corresponding interrupt can be enabled in the interrupt controller by setting the level control by InterruptLevel_u16. If the interrupt is called, the given callback function vector is executed after processing the essential issues of the UART / LIN driver, for further processing by the user application. If the function vector is *NULL*, then the callback is disabled.
Within the UART / LIN driver, the interrupt call is updating the communication status (ok or failure), which influences the further processing of transmission and reception requests of upper software layers.

The interrupt specification of InterruptSource_u08 is as follows:

<xxx>_IRQ_SEND:	Transmit interrupt
<xxx>_IRQ_RECEIVE:	Receive interrupt
<xxx>_IRQ_ERROR:	Error interrupt
<xxx>_IRQ_COMMON:	Shared interrupt for all above [(R3) only]

The interrupt level is a value as defined in the applied CPU core systems EI level interrupt control registers, using the *<xxx>_INTCLEAR #define* constant in the driver's mapping definition (see 3.4 for details and file information) to disable the interrupt. Use the *<xxx>_INTENABLEDEFAULT #define* constant by *OR* combination, in order to always set the *TBxxx* flag of the interrupt controller table reference mode - unless the vector table is mapped differently.

For each used UART / LIN controller unit, the following dedicated *#define* constants must be set in the driver's mapping definition (see 3.4 for details and file information):

#define <xxx>_INTM<*unit*>TX	*<devicefile register name of transmit interrupt>*
#define <xxx>_INTM<*unit*>RX	*<devicefile register name of receive interrupt>*
#define <xxx>_INTM<*unit*>ERR	*<devicefile register name of error interrupt>*
#define <xxx>_INTM<*unit*>COM	*<devicefile register name of common shared interrupt> [(R3) only]*

These settings will bind the interrupt controller registers to the UART / LIN controller unit *<unit>*; where each register is represented by its device file name.

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel [(R2) only]
InterruptSource_u08:	Selected interrupt source
InterruptLevel_u16:	Interrupt enable/level for interrupt controller setting
FunctionVector:	Pointer to a function returning void as a callback function

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

For each channel or unit of a LIN controller, each interrupt source can be selected by InterruptSource_u08 and their corresponding interrupt can be enabled in the interrupt controller by setting the level control by InterruptLevel_u16. If the interrupt is called, the given callback function vector is executed after processing the essential issues of the LIN driver, for further processing by the user application. If the function vector is *NULL*, then the callback is disabled.
Within the LIN driver, the interrupt call is updating the communication status (ok or failure), which influences the further processing of transmission and reception requests of upper software layers.

The interrupt specification of InterruptSource_u08 is as follows:

<xxx>_IRQ_SEND:	Transmit interrupt
<xxx>_IRQ_RECEIVE:	Receive interrupt
<xxx>_IRQ_ERROR:	Error interrupt

The interrupt level is a value as defined in the applied CPU core systems EI level interrupt control registers, using the <*xxx*>_*INTCLEAR #define* constant in the driver's mapping definition (see 3.4 for details and file information) to disable the interrupt. Use the <*xxx*>_*INTENABLEDEFAULT #define* constant by *OR* combination, in order to always set the *TBxxx* flag of the interrupt controller table reference mode - unless the vector table is mapped differently.

For each used LIN controller unit and channel, the following dedicated *#define* constants must be set in the driver's mapping definition (see 3.4 for details and file information):

#define <xxx>_INTM<*unit*>C<*channel*>TX	*<devicefile register name of transmit interrupt>*
#define <xxx>_INTM<*unit*>C<*channel*>RX	*<devicefile register name of receive interrupt>*
#define <xxx>_INTM<*unit*>C<*channel*>ERR	*<devicefile register name of error interrupt>*

These settings will bind the interrupt controller registers to the LIN controller unit <*unit*>, channel <*channel*>; where each register is represented by its device file name.

3.3.6.2 <xxx>_LINEnableInterrupt( )

Implementations: (R3).

(1) Implementations (R3)

(1-1) Parameters

UnitNumber_u08:	Selected UART / LIN Controller
Transmit_u08:	Transmit interrupt activation / deactivation
Receive_u08:	Receive interrupt activation / deactivation
ErrorGlobal_u08:	Global error interrupt activation / deactivation
ErrorBit_u08:	Bit error interrupt activation / deactivation
ErrorPhyBus_u08:	Physical bus error interrupt activation / deactivation
ErrorTimeout_u08:	LIN timeout error interrupt activation / deactivation
ErrorFrame_u08:	Framing error interrupt activation / deactivation
ErrorSync_u08:	*LIN SYNC field* error interrupt activation / deactivation
ErrorIDPar_u08:	*LIN ID parity* error interrupt activation / deactivation
TimeoutSel_u08:	LIN timeout error type selection: 0: Frame timeout, 1: Response timeout
Header_u08:	Header completion interrupt activation / deactivation

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Besides the function *<xxx>_CreateInterrupt( )*, where interrupts are activated in the interrupt controller, this function takes care of activation and deactivation of interrupt sources in the UART / LIN controller module. For activation of an interrupt, both the interrupt controller and the UART / LIN controller interrupt settings must be activated.
Error interrupts must be globally activated by *ErrorGlobal_u08*, details to be reported via interrupt can be chosen by the additional error interrupt sources as indicated.
Not all interrupts will be available in all operation modes. For example, a timeout error interrupt is not available when using UART mode. See details in the user's manual for this.

3.3.6.3 <xxx>_LINRestart( )

Implementations: (R2), (R3).

(1) Implementations (R2), (R3)

(1-1) Parameters

UnitNumber_u08:	Selected UART / LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel ((R2 only)

(1-2) Functional Description

Initiates a soft-reset of the LIN controller.
Used when switching to operation mode after a wake up event.

3.3.7 UART Controller Communication & Status

3.3.7.1 <xxx>_SendByte( )

Implementations: (A), (D), (E), (F), (R3).

(1) Implementations (A), (D), (E), (F), (R3)

(1-1) Parameters

UnitNumber_u08:	Selected UART Controller
SendByte_u08:	Data Byte to be sent

(1-2) Return Values

<xxx>_ERROR on parameter failures and failing TX verification when using *<xxx>_TP*, otherwise *<xxx>_TRANSFER_OK*.

(1-3) Functional Description

Initially after reset, the function is starting a transmission of the given *SendByte_u08* immediately. All subsequent calls of the function will have a verification phase at first, whether the last transmission was successfully completed.
The communication parameters for the transmission need to be set beforehand by calling the function *<xxx>_UARTModeConfig( )* or *<xxx>_Configuration( ), <xxx>_LegacyConfig( )*, respectively.
This verification is done by using the UART TX interrupt. If *UnitNumber_u08* is OR-ed with *<xxx>_TP*, then the UART TX interrupt is verified by checking the interrupt flag of the UART in the interrupt controller. Otherwise, a UART TX interrupt vector must have been activated (see *3.3.6.1 <xxx>_CreateInterrupt( )*), so that the driver is called upon TX completion.
If once the TX interrupt did not appear within *<xxx>_SENDTIMEOUT*, neither by vector, nor by flag in the interrupt controller, a *transmission failure* is noted, which in the following will switch the TX completion verification from checking the TX interrupt flag to the UART transmission in progress hardware flag. The user is informed about a transmission failure by the output of a '#' character.
Target of this approach is to allow a communication, even if some obstacles had happened like missing interrupt vector or missing interrupt flag in the interrupt controller.

3.3.7.2 <xxx>_ReceiveByte( )

Implementations: (A), (D), (E), (F), (R3).

(1) Implementations (A), (D), (E), (F), (R3)

(1-1) Parameters

UnitNumber_u08: Selected UART Controller
ReceiveByte_pu08: Data Byte received (passed by pointer reference)

(1-2) Return Values

<*xxx*>*_ERROR* on parameter failures or no reception ready to get,
otherwise <*xxx*>*_TRANSFER_OK*.

(1-3) Functional Description

If no *transmission failures* had happened within the <*xxx*>*_SendByte( )* function, then the reception status is taken from within the UART driver, where the called RX interrupt vector had set the valid reception flag, if data has been received since the last call. In case of a reception has been noted, the data byte is passed by the pointer reference.
If *transmission failures* had happened since the last reset within the <*xxx*>*_SendByte( )* function, then the reception status is taken either from hardware UART reception flag or the RX interrupt flag of the interrupt controller. If either one is set, then a reception is noted, and the data byte is passed by the pointer reference.
By this processing, the communication can be established even though some obstacles like missing interrupt vector or interrupt flag had happened.

3.3.7.3 <xxx>_GetStatus( )

Implementations: (A), (D), (E), (F).

(1) Implementations (A)

(1-1) Parameters

UnitNumber_u08:	Selected UART Controller
ParityError_pu08:	Parity error status (set on error, passed by pointer reference)
FrameError_pu08:	Frame error status (set on error, passed by pointer reference)
OverrunError_pu08:	Overrun error status (set on error, passed by pointer reference)
SendFlag_pu08:	Transmission ongoing (if set, passed by pointer reference)

(1-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(1-3) Functional Description

The function is reading the error status values as shown in the parameter list.
In addition, the transmission ongoing status is returned, and a dummy-read access on the data receive buffer is performed. This clears the reading buffer by wasting the data, restoring readiness to receive new data (clears the overrun status).

(2) Implementations (D), (E), (F)

(2-1) Parameters

UnitNumber_u08:	Selected UART Controller
SBFSuccessful_pu08:	LIN *header* successfully received indication (passed by pointer ref)
DataConsistencyError_pu08:	LIN *physical bus error* indication (passed by pointer reference)
ParityError_pu08:	Parity error status (set on error, passed by pointer reference)
FrameError_pu08:	Frame error status (set on error, passed by pointer reference)
OverrunError_pu08:	Overrun error status (set on error, passed by pointer reference)
SendFlag_pu08:	Transmission ongoing (if set, passed by pointer reference)

(2-2) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(2-3) Functional Description

The function is reading the error status values as shown in the parameter list.
For LIN support, the reception of a valid LIN header is indicated, and the LIN physical bus error is checked (LIN verifies its transmission by reading back the bus value, which must match).
In addition, the transmission ongoing status is returned, and a dummy-read access on the data receive buffer is performed. This clears the reading buffer by wasting the data, restoring readiness to receive new data (clears the overrun status).
All status flags are cleared by the function call, so that errors are cleared.

3.3.7.4 <xxx>_GetUARTStatus( )

Implementations: (R3).

(1) Implementations (R3)

(1-1) Parameters

UnitNumber_u08:	Selected UART Controller
BitError_pu08:	Bit error status (set on error, passed by pointer reference)
ParityError_pu08:	Parity error status (set on error, passed by pointer reference)
FrameError_pu08:	Frame error status (set on error, passed by pointer reference)
OverrunError_pu08:	Overrun error status (set on error, passed by pointer reference)
SendFlag_pu08:	Transmission ongoing (if set, passed by pointer reference)

(1-2) Return Values

<*xxx*>*_ERROR* on parameter failures, otherwise <*xxx*>*_OK*.

(1-3) Functional Description

The function is reading the error status values as shown in the parameter list.
In addition, the transmission ongoing status is returned, and a dummy-read access on the data receive buffer is performed. This clears the reading buffer by wasting the data, restoring readiness to receive new data (clears the overrun status).
All status flags are cleared by the function call, so that errors are cleared.

3.3.8 LIN Controller Communication & Status

3.3.8.1 <xxx>_SendBreak( )

Implementations: (A), (D), (E), (F).

(1) Implementations (A), (D), (E), (F)

(1-1) Parameters

UnitNumber_u08: Selected LIN Controller

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Transmits the *BREAK* signal to initiate a new LIN frame as LIN master.
The *SYNC* and subsequent LIN frame fields need to be transmitted as a data bytes with content 0x55 (see *<xxx>_SendByte( )*).

3.3.8.2 <xxx>_LINReceiveBreak( )

Implementations: (A), (D), (E), (F).

(1) Implementations (A), (D), (E), (F)

(1-1) Parameters

UnitNumber_u08: Selected LIN Controller

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

Triggers the reception of the *BREAK* signal.
When in LIN Master mode, call this function right before *<xxx>_SendBreak( )*. This will initiate the verification of the sent *BREAK* signal to verify the physical bus consistency.
In implementations (A), (D), (E) and (F) in fixed baud rate LIN slave mode, call this function to be ready to receive the next *BREAK* signal from the LIN master. In these implementations and mode, the LIN header decoding needs to be performed by software completely; so as next, the reception of the *SYNC* and *PID* fields shall be expected as received data bytes (RX interrupt).
In implementation (F) in auto baud rate LIN slave mode, don't use this function. This mode will trigger the RX interrupt implicitly, if a LIN header has been received and decoded (including *SYNC* and *PID* fields).

3.3.8.3 <xxx>_LINReadBreak( )

Implementations: (A), (D), (E), (F).

(1) Implementations (A), (D), (E), (F)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ReceiveBreakPending_pu08:	Set, if *BREAK* field was received (passed by pointer reference)
ReceiveBreakActive_pu08:	Set, if *BREAK* reception was triggered (passed by pointer reference)
SendBreakActive_pu08:	Set, if *BREAK* transmission was triggered (passed by pointer ref.)

(1-2) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(1-3) Functional Description

After triggering a *BREAK* reception or transmission, the status of this process can be checked by using this function. The function can be used to synchronize with the LIN frame protocol in implementations (A), (D), (E) and (F) in fixed baud rate mode.

3.3.8.4 <xxx>_LINSReceiveHeader( )

Implementations: (R3).

(1) Implementations (R3)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller

(1-2) Return Values

<xxx>_ERROR on parameter failures and in case of LIN controller inactive, otherwise *<xxx>_OK*.

(1-3) Functional Description

Triggers the reception of a LIN header on LIN slave (fixed and auto baud rate) implementations. Corresponds to function *<xxx>_LINReceiveBreak( )* in other implementations, but in (R3), this comprises the full LIN header, including *BREAK, SYNC* and *PID*.

3.3.8.5 <xxx>_LINS_ActivateResponses( )

Implementations: (F).

(1) Implementations (F)

(1-1) Parameters

Activation_bit:	Activation of LIN Slave

(1-2) Functional Description
Activates the LIN Slave to be ready to receive the first LIN header.

3.3.8.6 <xxx>_LINGetStatus( )

Implementations: (F), (R2), (R3).

(1) Implementations (F)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
HeaderReceived_pu08:	Valid LIN header reception, if set (passed by pointer reference)
DataReceived_pu08:	Valid LIN response completed, if set (passed by pointer reference)
PID_pu08:	*PID* field in case of header reception (passed by pointer reference)

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel
NoAction_pu08:	None of the following conditions are set (passed by pointer reference)
HeaderComplete_pu08:	Valid LIN header reception, if set (passed by pointer reference)
DataReceived_pu08:	Valid LIN response received, if set (passed by pointer reference)
DataTransmitted_pu08:	Valid LIN response transmitted, if set (passed by pointer reference)
PID_pu08:	*PID* field in case of header reception (passed by pointer reference)

(3) Implementations (R2)

(3-1) Parameters

UnitNumber_u08:	Selected LIN Controller
HeaderReceived_pu08:	Valid LIN header reception, if set (passed by pointer reference)
DataReceived_pu08:	Valid LIN response received, if set (passed by pointer reference)
DataTransmitted_pu08:	Valid LIN response transmitted, if set (passed by pointer reference)
PID_pu08:	*PID* field in case of header reception (passed by pointer reference)

(4) Implementations (F), (R2), (R3)

(4-1) Return Values

<*xxx*>*_ERROR* on parameter failures, otherwise <*xxx*>*_OK*.

(4-2) Functional Description

Yields the status of the LIN frame protocol, when processing the LIN frames.
Whenever a LIN header is received, the flag *HeaderReceived_pu08* (*HeaderComplete_pu08* in case of (R2)) is set. In this case, the parameter *PID_pu08* contains the actual *PID* of the frame.
Exclusively to that, a LIN response completion is indicated by the flags *DataReceived_pu08* and *DataTransmitted_pu08*, depending on the data direction of the LIN controller. In case of implementation (F), only the common flag *DataReceived_pu08* is provided for both directions.
Implementation (R2) in addition supports the *NoAction_pu08* flag, to improve processing in case of no action is yet required by software (processing is ongoing).

3.3.8.7 <xxx>_LINSendMessage( )

Implementations: (F), (R2), (R3).

(1) Implementations (F)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
PID_u08:	*PID* field in case of LIN master operation mode
MessageLength_u16:	Length of LIN response to be transmitted excluding checksum
EnhancedCheckSum_u16:	Specification of LIN 2.x checksum generation, if set
SoftCheckSum_u08:	Checksum to be transmitted in fixed baud rate or LIN master mode
OperationType_u08:	Either <xxx>_MODE_LINSLAVE or <xxx>_MODE_LINMASTER
Data_pu08:	Data value array (referenced by pointer) to be transmitted

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel
PID_u08:	*PID* field of LIN master frame generation
MessageLength_u08:	Length of LIN response excluding checksum
EnhancedCheckSum_u08:	Specification of LIN 2.x checksum generation, if set
Direction_u08:	Direction of response, either <xxx>_RESPONSE_TRANSMIT or <xxx>_RESPONSE_RECEIVE
Data_pu08:	Data value array (referenced by pointer)

(3) Implementations (R3)

(3-1) Parameters

UnitNumber_u08:	Selected LIN Controller
PID_u08:	*PID* field in case of LIN master operation mode
MessageLength_u08:	Length of LIN response excluding checksum
EnhancedCheckSum_u08:	Specification of LIN 2.x checksum generation, if set
Direction_u08:	Direction of response, either <xxx>_RESPONSE_TRANSMIT or <xxx>_RESPONSE_RECEIVE
Data_pu08:	Data value array (referenced by pointer)
OperationMode_pu08:	Currently active operation mode (passed by pointer reference)

(4) Implementations (F), (R2), (R3)

(4-1) Return Values

<xxx>_ERROR on parameter failures or on pending communication ((R2), (R3) implementations), otherwise *<xxx>_OK*.

(4-2) Functional Description

Initiates the transmission of LIN header and response (reception depending on direction) in LIN master mode. In LIN slave mode, response transmission or reception is triggered depending on direction setting.
(R2), (R3): Checksum is always calculated automatically, this also includes the *PID* parity.
(F): Depending on configuration, the frame *CRC* needs to be provided as *SoftCheckSum_u08* (LIN master and LIN slave fixed baud rate modes). The *PID* parity must be provided within the *PID_u08*.

3.3.8.8 <xxx>_LINReceiveAction( )

Implementations: (F).

(1) Implementations (F)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ActionType_u08:	Selection whether to receive the message or to skip it
MessageLength_u16:	Length of message to be received
EnhancedCheckSum_u16:	Specification of LIN 2.x checksum generation, if set

(1-2) Functional Description

To be used in LIN slave mode.
Initiates the reception (*ActionType_u08* set to <xxx>_A_GETMESSAGE) or ignores the message (*ActionType_u08* set to <xxx>_A_SKIPMESSAGE), which was announced by the previously received message header. The message header reception is active automatically after starting the LIN slave mode and calling the function *<xxx>_LINS_ActivateResponses( )* initially.

3.3.8.9 <xxx>_LINSReceiveAction( )

Implementations: (R3).

(1) Implementations (R3)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ActionType_u08:	Initiates, receives or skips a message
MessageLength_u16:	Length of message to be received
EnhancedCheckSum_u16:	Specification of LIN 2.x checksum generation, if set

(1-2) Functional Description

To be used in LIN slave mode.
Initiates the reception of a new message (*ActionType_u08* set to <xxx>A_GETHEADER), receives a message (*ActionType_u08* set to <xxx>_A_GETMESSAGE), or ignores the message (*ActionType_u08* set to <xxx>_A_SKIPMESSAGE), which was announced by the previously received message header. Reception of a new message header must be triggered explicitly by using this function, before any header can be recognized.

3.3.8.10 <xxx>_LINReceiveMessage

Implementations: (F), (R2), (R3).

(1) Implementations (F)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
MessageLength_u16:	Length of LIN response to be received excluding checksum
CheckSum_pu08:	Checksum received with this frame
Data_pu08:	Data value array (referenced by pointer) to be received

(2) Implementations (R2)

(2-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel
MessageLength_u08:	Length of LIN response to be received excluding checksum
Data_pu08:	Data value array (referenced by pointer) to be received

(3) Implementations (R3)

(3-1) Parameters

UnitNumber_u08:	Selected LIN Controller
MessageLength_u16:	Length of LIN response to be received excluding checksum
Data_pu08:	Data value array (referenced by pointer) to be received

(4) Implementations (F), (R2), (R3)

(4-1) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(4-2) Functional Description

A previously triggered reception of a message is fetched from the peripheral and copied to the array of data specified.
(R2), (R3): Checksum is always calculated automatically, this also includes the *PID* parity.
(F): The frame *CRC* (Checksum) needs to be verified by software.

3.3.9 LIN Controller Power Management

3.3.9.1 <xxx>_LINSendWakeup( )

Implementations: (R2), (R3).

(1) Implementations (R2)

(1-1) Parameters

UnitNumber_u08:	Selected LIN Controller
ChannelNumber_u08:	Selected LIN Controller channel
WakeupLength_u08:	Length of LIN wakeup signal in bits

(2) Implementations (R3)

(2-1) Parameters

UnitNumber_u08:	Selected LIN Controller
WakeupLength_u08:	Length of LIN wakeup signal in bits

(3) Implementations (R2), (R3)

(3-1) Return Values

<*xxx*>_*ERROR* on parameter failures, otherwise <*xxx*>_*OK*.

(3-2) Functional Description

Transmits a wakeup frame (signal) by the selected unit and channel, with the given length of bits.
To do this, the LIN controller is set to wakeup mode (<xxx>_MMODE_WAKEUP) with intermediate reset state (<xxx>_MMODE_RESET).
The function waits, until the wakeup frame has been transmitted. After that, the regular operation mode is restored (<xxx>_MMODE_OPERATION).

3.3.9.2 <xxx>_LINReceiveWakeup( )

Implementations: (R2), (R3).

(1) Implementations (R2)

(1-1) Parameters

UnitNumber_u08: Selected LIN Controller
ChannelNumber_u08: Selected LIN Controller channel

(2) Implementations (R3)

(2-1) Parameters

UnitNumber_u08: Selected LIN Controller

(3) Implementations (R2), (R3)

(3-1) Return Values

<xxx>_ERROR on parameter failures, otherwise *<xxx>_OK*.

(3-2) Functional Description

Initiates the trigger to wait on a wakeup frame (signal) by the selected unit and channel.
To do this, the LIN controller is set to wakeup mode (<xxx>_MMODE_WAKEUP) with intermediate reset state (<xxx>_MMODE_RESET).
After that, the function returns. By software (wakeup interrupt), the wakeup condition must be verified and the LIN controller restarted by calling *<xxxx>_LINRestart( )*.

3.4 Mapping of the Lower LIN / UART Driver

3.4.1 Device Level

On device level, the type and amount of LIN/UART controllers are defined. Depending on the implemented CAN controller type, the following two entries in the file *map_device.h* are set:

#define <xxx>_MACROS	(*n*)	// Number of LIN/UART controllers
#define <xxx>_TYPE	(*m*)	// Type of LIN/UART controllers

Within the driver package, these lines are already filled in and do not require any change. For information, the available LIN/UART controller types are listed in the common resource file *ree_macros.h*, which is also included in the driver packages.

3.4.2 LIN / UART Controller IP Level

On IP level, device specific properties of the implemented CAN controller type(s) are defined in the files *map_*.h*. The file name depends on the CAN controller type:

<xxx> is ...	UARTA:	map_uarta.h
	UARTD:	map_uartd.h
	UARTE:	map_uarte.h
	UARTF:	map_uartf.h
	RLIN2:	map_rlin2.h
	RLIN3:	map_rlin3.h

Within the *map_*.h* files, the following subsection will explain the various settings, and how to adapt them to individual needs. It is not recommended to change any other parameters of these files, which are not documented below.

3.4.2.1 Base Addresses

It is not recommended to change the base addresses. These are defining how and where the LIN/UART controller and also some functional sub-parts are mapped within the device memory. The following entries can be found:

#define <xxx>_BASE
#define <xxx>_OFFSET_<...>

3.4.2.2 Device and Usage Adaptation

Several parameters exist, which may be altered by the user. Not all implementations are supporting every parameter.

(1) <xxx>_FREQFACTOR
This entry defines the factor between the given system clock frequency *OSCILLATOR_FREQUENCY* and the communication clock of the LIN/UART controller. Typically, the factor is 0.5 for most implementations. If a PLL is enabled, its frequency should be considered within the *OSCILLATOR_FREQUENCY* setting. The settings for *OSCILLATOR_FREQUENCY* are located in the configuration file *map_asmn_basic.h* (for the serial monitor program) or *map_tgmn.h* (for the graphics monitor program) respectively.

(2) <xxx>_MAXBAUDRATE
<xxx>_MINBAUDRATE
Defines the maximum and minimum selectable bit rates. This depends on design capabilities of the controller and its assoicated ports, and on the maximum possible clock division factor of the unit.

(3) <xxx>_SENDTIMEOUT
The value indicates the amount of idling loops, while waiting on hardware reactions. This timeout supervision is mostly used when starting or shutting down, or when accessing common resources within a LIN/UART controller, which are protected by semaphore mechanisms. Depending on the speed of the CPU, it may be required to adjust the value.

(4) <xxx>_PORT ...
Used to set the port I/O properties of the CAN controller and all of its channels. Redirection to dedicated pins and ports is done by setting these constants. See *3.3.4.1 <xxx>_PortEnable( )* for details.

(5) <xxx>_INT_BUNDLINGHOOK
If a LIN/UART controller provides internal bundling of interrupt sources, so that less resources are required by the interrupt controller, then this variable is set to an additional support function, which is device specific and decodes the bundled interrupts. The variable can be used to insert additional interrupt execution.

(6) <xxx>_INT ...
Used to associate interrupt sources of a CAN controller with interrupt controller registers, which are used to enable or disable an interrupt. See *3.3.6.1 <xxx>_CreateInterrupt( )* for details.

(7) <xxx>_INTCLEAR
Constant to write into interrupt controller registers, if an interrupt shall be disabled. See the CPU core manual for proper values.

(8) <xxx>_INTENABLEDEFAULT
Constant to write into interrupt controller registers, if an interrupt shall be enabled. Not used by the lower driver by itself, but by application examples. See the CPU core manual for proper values.

(9) <xxx>_INTFLAG
Bit mask of the corresponding interrupt controller register, which filters the interrupt flag.

3.4.2.3 Memory Vectors

In the last section of the *map*.h* file, the base addresses are applied to structures and arrays, in order to provide register access to the hardware for the low level driver.
These *static struct <xxx>_...* entries are specific for each LIN/UART controller type and must not be altered by the user. Configuration and tailoring of the specific implementation to the device is done here, too (i.e., amount of channels, number of units).

3.5 Applications Based on the Lower LIN / UART Driver

3.5.1 Serial Monitor Program

The serial monitor program is allowing the interactive execution of the application examples, as described in chapter *3.5.3 LIN Communication Application Examples*, by commands from a TTY terminal. If supplied with the software package, the monitor program resides in the files *asmn*.c*.

Depending on implemented serial interfaces and the user's preference, the command console can be executed using certain resources. Within the file *map_asmn_basic.h*, the command console is directed to a resource by the setting of *ASMN_UARTINTERFACE*. The following UART types are compatible with the application examples (availability depends on the device):

V850 series:
UARTA_STANDARD, UARTC_STANDARD, UARTD_STANDARD, UARTE_STANDARD

78K0R, RL78 series:
UARTF_STANDARD

RH850 series:
RLIN3_STANDARD

All series with debug system only (using the debugger terminal console):
DEBUG_INTERNAL

3.5.1.1 Using the Debugger Console

If the debugger terminal console shall be used, besides the setting above, the following settings shall be made in the configuration file *map_asmn.h*:

```
#define ASMN_MENUCOM                    ( 0 )
#define ASMN_UARTTRANSFEROK             ( true )
#define ASMN_UARTERROR                  ( false )
#define ASMN_MENUCOM_EXE1               ( 0x0D )
#define ASMN_MENUCOM_EXE2               ( 0x0A )
#define ASMN_MENUCOM_BUFLEN             ( 4 )
#define ASMN_UARTMODECRLF               ( 1 )
#define ASMN_UARTMODECR                 ( 0 )

#define ASMN_UARTMODEFORCELF

#define ASMN_UARTSENDSTRING             ASMN_SendString
#define ASMN_UARTSENDBYTE               ASMN_SendByte
#define ASMN_UARTRECEIVEBYTE            ASMN_ReceiveByte
#define ASMN_UARTRECEIVEINT             ASMN_ReceiveInteger
#define ASMN_UARTRECEIVELONG            ASMN_ReceiveULong
#define ASMN_UARTRECEIVEFLOAT           ASMN_ReceiveFloat
```

This will redirect all UART communication functions to internal functions of the monitor program.

3.5.1.2 Using a Serial Interface

Depending on the serial interface specified in *map_asmn_basic.h*, the corresponding low level driver of the interface and its device specific mappings have to be added to the project.

A *map*.h* file also belongs to the low level driver of the serial interface. This file has to be configured similarly like the mapping file of the low level CAN driver. Here, the appropriate port and interrupt associations have to be set, so that it fits with the hardware environment. As an example, for a serial interface called *RLIN3*, the mapping file *map_rlin3.h* needs to be configured properly. See the mapping of the CAN driver in *3.4 Mapping of the Lower LIN / UART Driver* section to have a generic overview.

In addition, the file *map_asmn.h* contains the serial interface parameters.

#define ASMN_MENUCOM	RLIN3_0	Serial Unit (here: *RLIN30*)
#define ASMN_MENUCOM_ILEVEL	0	Interrupt Level
#define ASMN_MENUCOM_BAUD	9600L	Bit rate
#define ASMN_MENUCOM_PARITY	RLIN3_PARITY_NONE	Parity setting
#define ASMN_MENUCOM_CHLEN	RLIN3_CHARLEN_8BITS	Bits per frame
#define ASMN_MENUCOM_STOP	RLIN3_ONESTOPBIT	Stop bits per frame
#define ASMN_MENUCOM_DIR	RLIN3_DIR_LSBFIRST	Bit ordering
#define ASMN_MENUCOM_EXE1	0x0D	First code for <return>
#define ASMN_MENUCOM_EXE2	0x0A	Alternative code
#define ASMN_MENUCOM_BUFLEN	4	Command buffer size

The monitor calls to initiate the user interface dialogue are mapped to the low level driver functions of the used serial interface. This is done with the set of *ASMN_UART...* definitions. When changing the console, the new driver functions can be entered here.

3.5.2 Graphics Monitor Program

For dedicated hardware of Renesas product series on certain application boards made by Renesas, where a graphics display with touchscreen can be installed, there is a graphics monitor program available for demonstration purposes.

The graphics monitor program is allowing the interactive execution of the application examples, as described in chapter *3.5.3 LIN Communication Application Examples*, by commands from a touchscreen. If supplied with the software package, the monitor program resides in the files *tgmn*.c*.

Within the file *map_tgmn.h*, the graphics monitor program is configured regarding its underlying graphics display routines. It includes libraries from *zlib* and *libpng*, which are licensed under the premises of the *Open Source Initiative* and *Free Software Foundation*. Corresponding disclaimers and license text can be found in *3.5.2.1 Public Licenses of Graphics Routines*.

The graphics monitor program requires an additional hardware dependent support package, which is located in the files "*bsp_tgmn.**", where settings of physical connections and required additional resources (timer) are defined.

Using the structures in the files "*tgmn_<xxx>_tgmnif.h*", the application example functions are called by vectors and parameters gathered from the graphical user interface.

Using the timer activated in the hardware dependent support package, the touchscreen is checked for any input events of the GUI, and any pending GUI updates are executed.

3.5.2.1 Public Licenses of Graphics Routines

(1) zlib

Copyright (c) 1995~2017 Jean-Loup Gailly, Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

(1-1) The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
(1-2) Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
(1-3) This notice may not be removed or altered from any source distribution.

(2) libpng

Copyright (c) 1996~2017 Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson et al.

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

(2-1) The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
(2-2) Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
(2-3) This notice may not be removed or altered from any source distribution.

3.5.3 LIN Communication Application Examples

The application examples of the LIN/UART driver reside in the files *_a.c* with API **_a.h*. The functions are called by the monitor program - so that the parameters can be entered via the terminal console.

Apart from the elementary data input and output functions (which are not described within this documentation), this part focuses on the LIN application driver, separated by LIN master and LIN slave functionality.

• Note that the (F) implementation only supports one LIN communication channel at a time.

3.5.3.1 LIN Communication Tables

For LIN master and LIN slave, lookup tables are required to allow their operation. These tables are defined and initially set up in the file **_s.h*.
Every table is based on the same structure, which is used in a 2-dimensional array for a LIN master (to allow switching of tables), or in a 1-dimensional array for a LIN slave. The arrays which build the tables are constrained by the following constants:

<xxx>_A_LIN_MASTERTABLES:	Number of LIN master tables
<xxx>_A_LIN_MAXTABLELENGTH:	Maximum size of a LIN master or slave table
<xxx>_A_LIN_MAXDATA:	Maximum length of a LIN message

The LIN table structure element "*<xxx>_a_lintable_entry*" consists of the following components; thus every structure element defines one LIN message and an execution information.

Type_u08:	*Execution Information* of the table
PID_u08:	Protected ID of the message (without parities set)
DataLength_u08:	Length of message
Data[<xxx>_A_LIN_MAXDATA]:	The data content of the message

The *Execution Information* of the table determines details on the handling and content of the message, and may have the following settings:

• <xxx>_A_TX:
The "*Data*" field contains data to be sent out, by master after the header, or by slave after receiving a header, which matches the *PID_u08*. The length of the response to be sent is determined by the *DataLength_u08* field. A slave checks the *PID_u08* against all table entries of this kind, to determine whether to send the response.

• <xxx>_A_RX:
The "*Data"* field is a container to be filled with data from a LIN response, as received from the master (in slave operation) or from an addressed slave (in master operation). Slaves are addressed in master mode by sending the *PID_u08* in the header. A slave checks the *PID_u08* against all table entries of this kind, to determine whether to receive the response.

• <xxx>_A_NO:
This table entry indicates the end of the table; all other field of this entry are not considered. The entry is mandatory to be placed as last entry of the table, and only one of this kind is allowed, so that a master will recognize a new table schedule or a slave recognizes the limit for its search PID lookup operation.
In addition, this entry contains the frame data of a sleep command frame. This frame will be sent on LIN master sleep transition.

3.5.3.2 LIN Application Layer Elementary Functions

The following steps are executed by additional functions of the "*_a.c" application layer, with API "*_a.h", to activate and control the LIN master or LIN slave function:

(1)	Set the LIN operation mode:	<xxx>_LIN_SetMode(...)	
(2)	Activate the LIN slave:	<xxx>_LINS_ActivateResponses(...)	* LIN slave only
(3)	Set the LIN master table:	<xxx>_LINM_SetTable(...)	* LIN master only
(4)	Start the LIN master table:	<xxx>_LINM_RestartTable(...)	* LIN master only
(5)	Wake up the LIN bus:	<xxx>_LINM_SendWakeup(...)	* LIN master only
(6)	Send (first) entry of the master table:	<xxx>_LINM_NextTableContent(...)	* LIN master only

After these steps, the LIN slave continues running automatically, driven by interrupts.
In implementations (R2) and (R3), the function <xxx>_LINSlaveActivation(...) is providing all steps to start a LIN Slave.

The LIN master continues with the next table entry, by doing the following steps:

(1)	Poll to check for the idle LIN bus:	<xxx>_LINM_GetStatus(...)
(2)	Send next entry of the master table:	<xxx>_LINM_NextTableContent(...)

In implementations (R2) and (R3), the function <xxx>_LINMasterExecution(...) is providing all steps and further processing to run a LIN Master.

3.5.3.3 Additional Application Layer Functions

In order to set or get response data for an upper layer application, the following functions are supported:

• <xxx>_LINS_SetTableContent(...)	Set LIN response data of a slave table entry.
• <xxx>_LINS_SetTablePID(...)	Set LIN response PID of a slave table entry.
• <xxx>_LINM_SetTableContent(...)	Set LIN response data of an entry of the current master table.
• <xxx>_LINS_GetTableContent(...)	Read LIN response data of a slave table entry.
• <xxx>_LINM_GetTableContent(...)	Read LIN response data of an entry of the current master table.

The following functions are used to handle LIN bus exceptions:

• <xxx>_LINM_ResetFrame(...)	Resets the LIN master in idle state, after an error.
• <xxx>_LINS_GetClearErrorState(...)	Resets the LIN slave in idle state, after an error.
• <xxx>_LINS_GetStatus(...)	Reads the LIN slave status (see states: 3.5.3.4), and restarts the LIN slave in case of wake-up or error.

The following functions are used to handle the LIN wake-up and sleep procedures:

• <xxx>_LINM_SetSleepState(...)	Sends the LIN sleep frame of the current master table and enters sleep mode, ready to receive a wake-up frame.
• <xxx>_LINS_SetSleepState(...)	Enters sleep mode of the LIN slave, ready to receive a wake-up frame.
• <xxx>_LINM_SendWakeup(...)	Sends the wake-up frame pattern, when in idle state.
• <xxx>_LINS_SendWakeup(...)	Sends the wake-up frame pattern, when in idle state, and restarts the LIN slave to receive a new header.
• <xxx>_LINS_GetSleepState(...)	Returns the LIN sleep state condition (see states: 3.5.3.4).

3.5.3.4 LIN Application Layer States

The following states are supported by the LIN application layer within "*_a.c*":

- <xxx>_A_LIN_STATE_IDLE:
  The application layer is idle, i.e., no LIN frame is pending to be handled or on the bus.
  In LIN master mode, the master waits for a software call to send the next frame of the current table.
  In LIN slave mode, the slave waits for the detection of a LIN header, which will cause an interrupt.

- <xxx>_A_LIN_STATE_NEWSDATA:
  Same as <xxx>_A_LIN_STATE_IDLE, but indicating that recently the slave receive response table part has been updated by a new master frame. Not supported by all implementations.

- <xxx>_A_LIN_STATE_BUSY:
  Global state during activity, this state is a mask for the following detail states:

  - <xxx>_A_LIN_STATE_TXBREAK:
    In LIN master mode, a new header transmission has been started.
  - <xxx>_A_LIN_STATE_RXDATA:
    Ready to receive a response. In implementations (R2) and (R3), the LIN master mode will not show this state, because the hardware is handling the full frame automatically.
  - <xxx>_A_LIN_STATE_TXDATA:
    Ready to transmit a response. In implementations (R2) and (R3), the LIN master mode will not show this state, because the hardware is handling the full frame automatically.
  - <xxx>_A_LIN_STATE_BRKDEL:
    Intermediate state after BREAK transmission and header fields of the LIN master. In implementations (R2) and (R3), the LIN master mode will not show this state, because the hardware is handling the full frame automatically.
  - <xxx>_A_LIN_STATE_RXSLEEP:
    Ready to verify the sleep message content in the LIN slave, after getting its PID from the master.

- <xxx>_A_LIN_STATE_SLEEP:
  Entered after sending the sleep command frame by using *<xxx>_LIN*SetSleepState* or after receiving a message which matches the sleep frame table entry, marked with "*<xxx>_A_NO*".

- <xxx>_A_LIN_STATE_WAKEUP:
  Entered after sending a wake-up frame by using *<xxx>_LIN*SendWakeup* or after receiving a wake-up frame, when in sleep state.

- <xxx>_A_LIN_STATE_ERROR:
  Global state of error, this state is a mask for the following errors in detail. In case of errors, any current LIN frame will be discarded before reception or its transmission will be aborted.

  - <xxx>_A_LIN_STATE_FSMERROR:
    State is entered, if the LIN protocol is heavily disturbed or in disorder, which does not allow a processing.
  - <xxx>_A_LIN_STATE_UNEXRECV:
    State is entered, if the LIN protocol is heavily disturbed or in disorder while receiving a response, which does not allow a processing.
  - <xxx>_A_LIN_STATE_DLLERROR:
    State is entered, if the LIN protocol is heavily disturbed or in disorder while transmitting a response, which does not allow a processing, or on other LIN errors except checksum or data consistency.
  - <xxx>_A_LIN_STATE_DCCERROR:
    Data consistency check has failed during transmission; i.e., the LIN bus is not following the transmission attempt. Check transceiver or physical bus to fix it.
  - <xxx>_A_LIN_STATE_PIDERROR:
    Parity error of the protected ID (PID) was found. Check the compatibility with the selected protocol version.

- <xxx>_A_LIN_STATE_RPRERROR:
  Indicates a "*Response Preparation Error*", which is available on implementations (R2) and (R3) only.
  Indicates, that software performance of the system is not enough to handle a frame in slave mode in time.
- <xxx>_A_LIN_STATE_CHSERROR:
  Checksum error of the received frame response.

3.5.3.5 LIN Master State Engine Description of UARTF

In implementation (F), the LIN Master State Engine looks as follows:

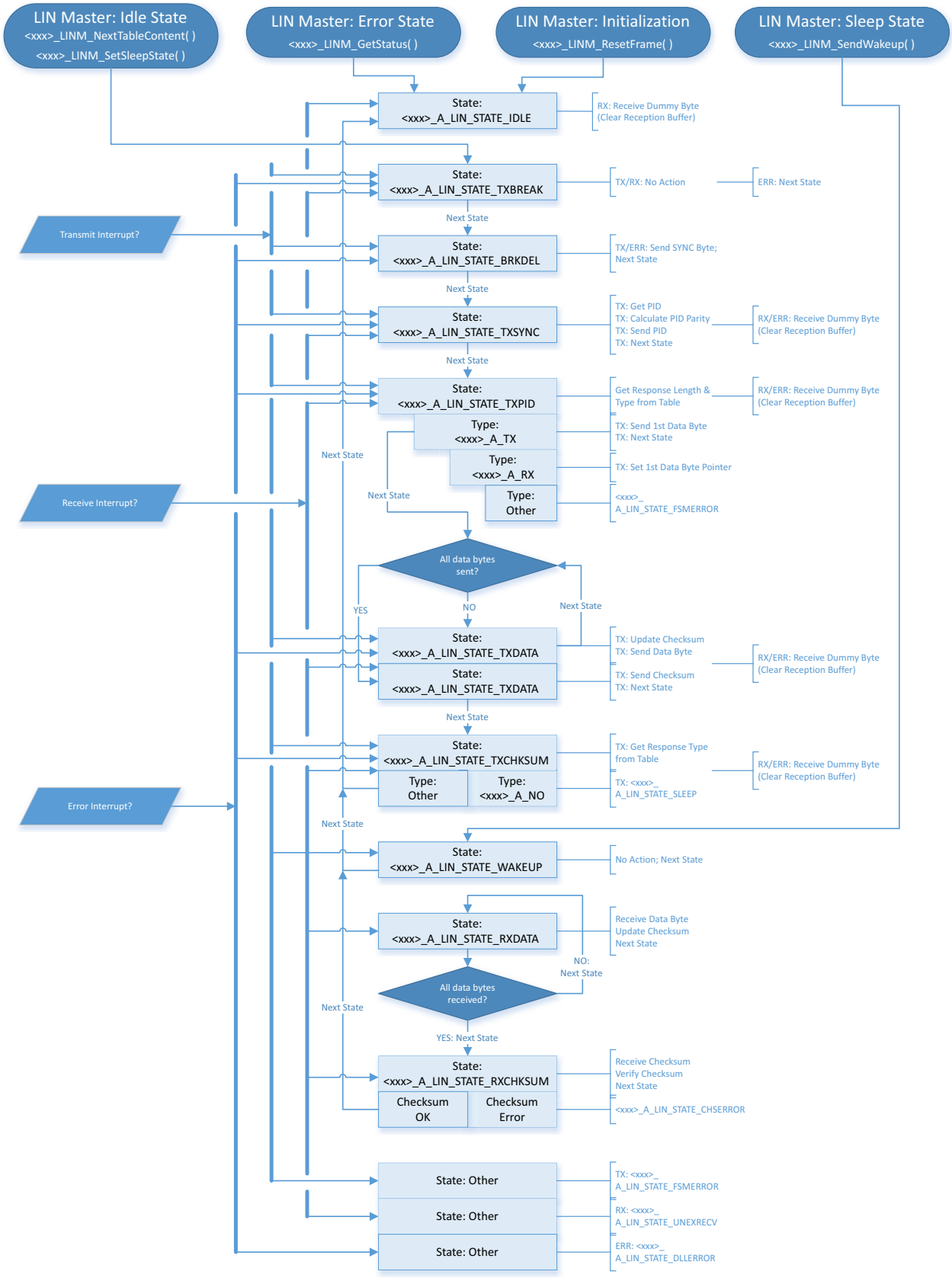

Figure 3.1 LIN Master State Engine of UARTF

3.5.3.6 LIN Slave State Engine Description of UARTF

In implementation (F), the LIN Slave State Engine looks as follows:

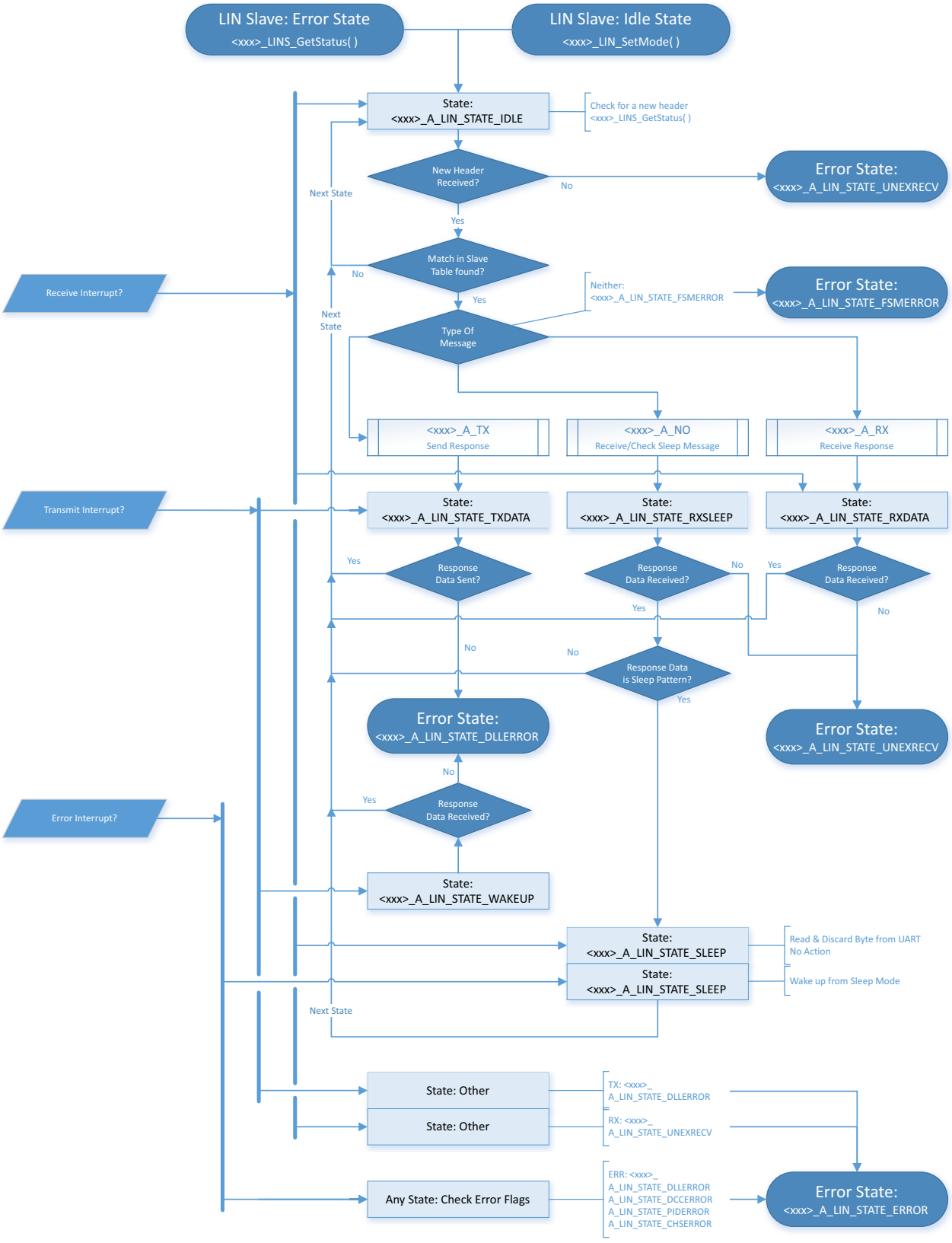

Figure 3.2 LIN Slave State Engine of UARTF

3.5.3.7 LIN Master State Engine Description of RLIN2, RLIN3

In implementations (R2) and (R3), the LIN Master State Engine looks as follows:

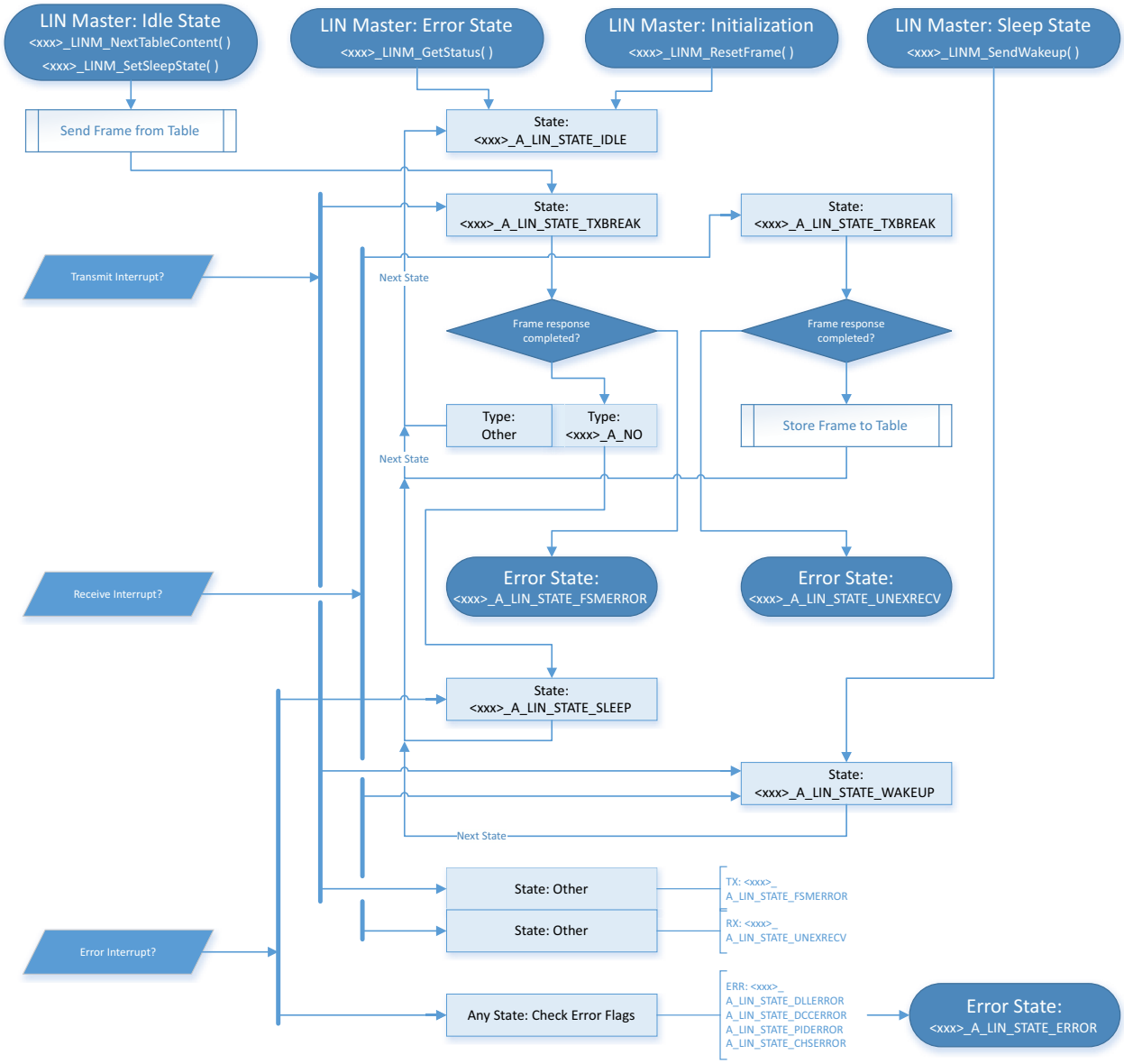

Figure 3.3 LIN Master State Engine of RLIN2 and RLIN3

3.5.3.8 LIN Slave State Engine Description of RLIN3

In implementation (R3), the LIN Slave State Engine looks as follows:

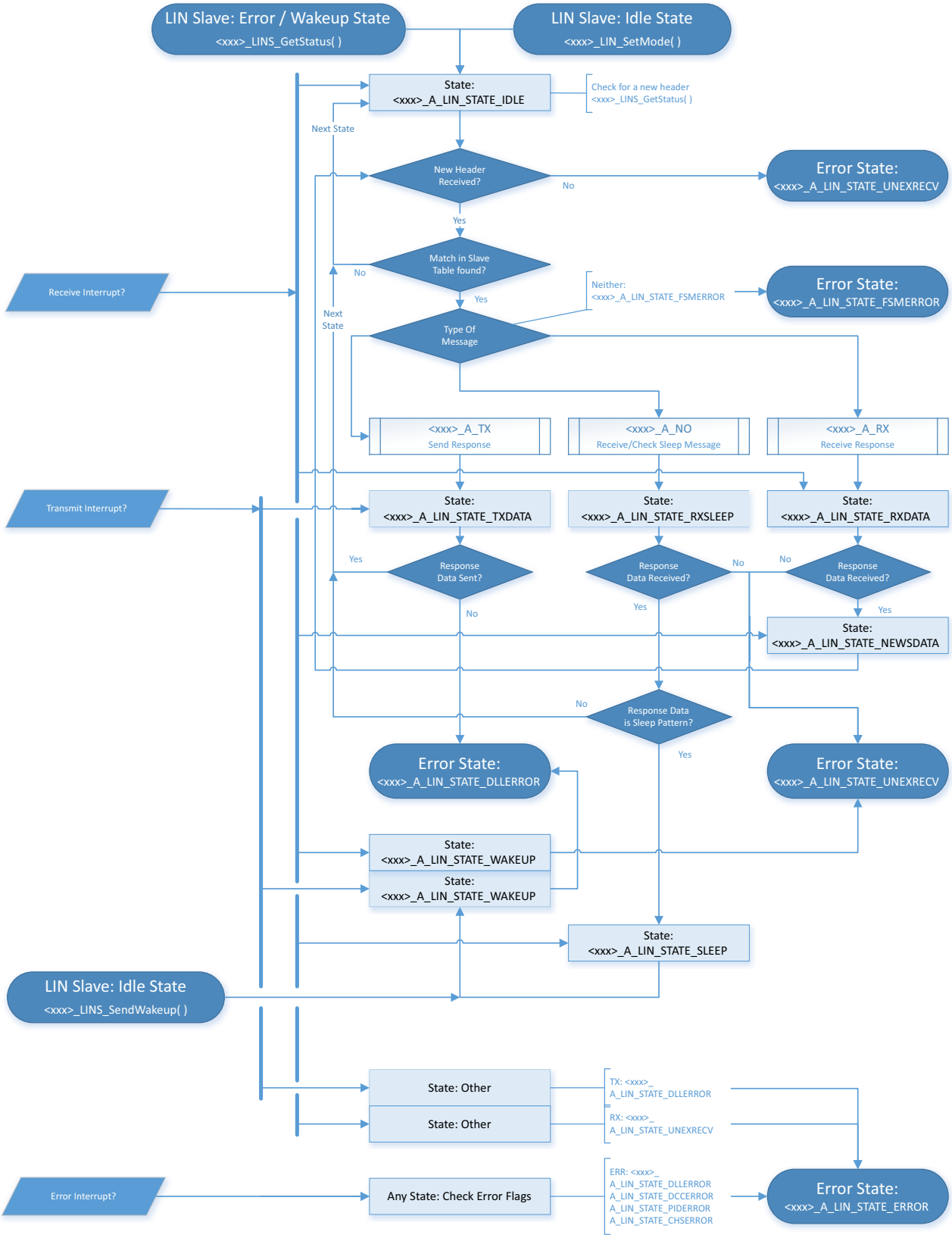

Figure 3.4 LIN Slave State Engine of RLIN3

4. Frequently Asked Questions

4.1 Interrupts

4.1.1 Interrupt Handling in RL78 RLIN3 Implementations

The interrupt controller in RL78 is triggered by edges of interrupt indications of peripherals, and so for RLIN3, too. On the other hand, the interrupt sources of RLIN3 are level based.
For this reason, when handling RLIN3 interrupts in RL78, all interrupt sources within RLIN3, which are sharing the same interrupt flag of RL78 must be handled and cleared, as soon as the interrupt is executed by RL78.
As an overview, the following interrupt sources of RLIN3 are grouped in RL78:

Table 4.1 Shared RL78 Interrupt Sources of RLIN3

Interrupt Source		Shared Interrupt Events	Operation Modes	Interrupt Event Flags to Clear
Common	Separated			
LIN	Transmission	-	all	-
	Reception	-	all	-
	Status	Bit Error	all	BER in LEST Register
		Physical Bus Error	LIN Master	PBER in LEST Register
		Frame / Response Timeout Error	LIN	(F)TER in LEST Register
		Framing Error	ALL	FER in LEST Register
		Checksum Error	LIN	CSER in LEST Register
		Response Preparation Error	LIN	RPER in LEST Register
		Sync Field Error	LIN Slave	SFER in LEST Register
		ID Parity Error	LIN Slave	IPER in LEST Register
Not applicable		Overrun Error	UART	OER in LEST Register
		Expansion Bit Detection	UART	EXBT in LEST Register
		ID Match	UART	IDMT in LEST Register
		Parity Error	UART	UPER in LEST Register

Note: As an example, when using the common LIN interrupt in LIN Slave mode, the flags BER, (F)TER, FER, CSER, RPER, SFER and IPER must always be checked and cleared if set, as soon as the common LIN interrupt has been handled. Otherwise, no further common LIN interrupt would be generated. The common LIN interrupt is not available in UART operation mode; therefore the flags OER, EXBT, IDMT and UPER need not to be checked in LIN Slave mode.
As a secondary example, when using the separated transmission interrupt in either operation mode, no additional flag needs to be cleared during interrupt handling.

4.2 Status Information

4.2.1 Status Information of LMA Implementations

The LMA unit is able to provide additional status information, while it is performing steps of the LIN protocol execution. This can be read by software to see and verify the LIN protocol execution. The information is available by reading the SSL register bits.

Table 4.2 SSL Register Detail Information

SSL Content	LIN Protocol Context
0x00	Initial State
0x18 / 0x1B / 0x1C	UART write access for BF transmission demands
0x1D	Waiting for BF transmission start (UART interrupt)
0x24, 0x25, 0x26	UART write access for status clear
0x10, 0x11, 0x12	UART write access for status clear
0x21, 0x22	UART write access for SF transmission demands
0x23	Waiting for BF transmission completion (UART interrupt)

4.3 Data Reception

4.3.1 Receiving UART Data after an Error Interrupt (all IP types)

Whenever an Error Interrupt has been received instead of a Reception Interrupt, the following actions have to be taken by the application software:

- Check for all error flags
- Clear all error flags
- Read the reception data buffer register once and discard the data value

Revision History	LIN / UART Controller Application Note

Rev.	Date	Description	
		Page	Summary
01.00	March 2015	—	First Edition issued
01.01	March 2015	3	FAQ section updated
02.00	February 2019	3, 4	LIN Sample Software description inserted as Chapter 3. Topics added to FAQ section.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)
   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on
   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state
   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins
   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals
   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin
   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses
   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products
   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. 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 or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving 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, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, 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 and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall 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. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, 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.

(Note1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

© 2019 Renesas Electronics Corporation. All rights reserved.