LIN Slave Driver
How to Get Started

Changes	Date	Name	Short
0.1 – Initial draft	2013-05-06	Barbara Rögl	BR
1.0 – Initial release	2013-05-29	Barbara Rögl	BR
1.1 – Corrected cycle time for ISR routine in polling mode, removed HW reference	2013-11-21	Barbara Rögl	BR
1.2 – Added Quick Start Instructions	2015-08-28	Paul Park / Barbara Rögl	PP & BR

ihr **GmbH**
Airport Boulevard B210
77836 Rheinmuenster

Tel.: +49 7229 / 18475-0
Fax: +49 7229 / 18475-11
homepage: http://www.ihr.de
e-mail: ihr-online@ihr.de

Author	Barbara Rögl
Version	1.2

1 References

No.	Document/Description	Rev./ Date
1	LIN Driver API Description M&S v1.1 2010-10-28.pdf	1.1
2	User Guide – LDC-Tool.pdf	2.0.x.x
3	Release Notes – LDC-Tool.pdf	2.0.x.x
4	LIN Specification 1.3	1.3
5	LIN Specification 2.0	2.0
6	LIN Specification 2.1	2.1

2 LIN Details

LIN Version supported	1.3, 2.0 (tested), 2.1 (tested)
Diagnostic Api	RAW API (LIN 2.0, 2.1), Cooked (LIN 2.0, 2.1)
Baudrate	2400, 9600, 10417, 19200 (tested), 20000 bps

3 Quick Start Step-by-Step Instructions

3.1 Download LDS Package from *ihr* Server.

3.2 Extract LDS Package & Verify Contents

1. LDS-Pack: Demo Application (main.c and main.h) and LIN Driver stack (/Protocol and /Hal folder or for older packages /LIN_DRIVER)
2. LDC-Tool: LIN Configuration generation tool (generates configuration files genLinConfig.c and .h)
3. Documents: All documentation for LIN Driver Package

3.3 Compile Demo Application Project

Verify that the Demo Application provided with LIN Driver package compiles using the IDE and Compiler installed on the user's machine.

3.4 Generate New Configuration Files (LDC-Tool)

Using the LDC-Tool, create a new set of configuration files (genLinConfig.c and genLinConfig.h) using the application-specific LIN Description File (LDF). For further information on using the LDC-Tool, please consult [2].

3.5 Copy Configuration Files into LIN Stack Folder

Copy the genLinConfig.c and genLinConfig.h files from LDC-Tool into LIN_Driver folder, overwriting the configuration files from the Demo Application.

3.6 Copy LIN Stack into Existing Project

Copy the LIN Stack with updated configuration files (/Protocol and /Hal folder or for older packages /LIN_DRIVER) into the Main Application project.

3.7 Implement Initialization of I/O, Interrupts, LIN Task Cycle

1. Setup UART RX/TX pins as Input/Output pins respectively to make sure there is no bus disturbance while the UART peripheral is not yet initialized or when it does not control the pins during re-initialization.
2. Define UART Interrupt Service Routine, and call l_ifc_rx() routine inside UART ISR.
3. Call l_cyclic_com_task() routine inside UART ISR. Alternatively, setup fast task cycle (check the datasheet for more detailed information on timing requirements).
4. Setup LIN Task Cycle as defined in LDC-Tool "Operation Mode" and call ld_task() routine inside LIN Task Cycle.
5. Verify that the LIN tasks are protected from IRQs as necessary (check the datasheet for more detailed information or "AN001 LIN Driver Integration – Interrupt Protection V1.0.pdf"

Please use Demo Application provided (main.c & main.h) as a reference when implementing these functions into user's Main Application.

4 LDC-Tool Configuration

The first step in a new project is to choose a new LDF (LIN Definition File) and generated the driver stack accordingly with the files generated by the LDC-Tool. For further information how to use the LDC-Tool please see [2].

The two files have to be put in the correct build directories so that the compiler is able to find all necessary files.

5 Necessary Callback functions

The user's application has to implement several calls that the API requires.

5.1 l_irqmask l_sys_irq_disable (void)

This function will activate or deactivate the interrupt of the UART / SCI so that the data can be fetched from or written to registers without interference from possible interrupts. This function is used within the driver. Usually the LIN is slow enough that the data can be fetched without filling the call, the latency for calling the ISR is usually less that one byte.

5.2 void l_sys_irq_restore (l_irqmask mask)

This function will reactivate the interrupts after reading from / writing to registers.

5.3 void LIN_Enable_Transceiver (void)

This function will enable the transceiver if the transceiver is to be enabled via a pin externally during startup initialization. If the user's application does not wish that the enable is set during initialization routines, this function has to be left blank.

5.4 void LIN_Disable_Transceiver (void)

This function will disable the transceiver if the transceiver is to be enabled via a pin externally. However this function is never called within the driver. Going to sleep and waking up should be done in the application.

6 Necessary calls in application

The driver itself consists of three tasks that have to be embedded into the application and one initialization routine.

6.1 l_ifc_rx()

This is the ISR of the driver. This function will check the state of the UART / SCI registers and react accordingly. It will also fetch the data from the receive register and put it into a temporary buffer to process later by a cyclic call.
The interval this function has to be called in polling mode varies depending on the UART / SCI of the device. If the UART / SCI has a buffer of at least one byte available the function has to be called once each byte. If the UART /SCI has no such buffer the function has to be called once per bit time.

6.2 l_cyclic_com_task()

This is the fast cyclic task. It will process the data fetched by the l_ifc_rx() function and therefore has only workload if data was received. It can be called immediately after the l_ifc_rx() in the ISR (if interrupt mode is used) if the interrupt load and latency for other interrupts with lower priority is allowing this. Another method would be to call it cyclically in the application, when used this way, it has to be called at least once each byte-time. It does not matter if it is called more frequently, because the workload is only generated by data marked for processing by the ISR.

6.3 ld_task()

This task has two major responsibilities:

- It is used as time base of the driver to calculate timeouts. The cyclic time is configured in the LDC-Tool. This includes timeout when using baud rate synchronization, timeout if a frame was not finished within a very generous timeslot (usually a timeout is recognized after not receiving a byte for 10ms), diagnostic messages timeout.

- It also handles the diagnostic messages. The configuration messages are handled directly by the configuration layer within the driver. When using user defined diagnostics, cooked or raw API all other diagnostic requests are forwarded to the application. Please see chapter diagnostics for further information.

6.4 l_sys_init()

This function will initialize the driver and UART / SCI for LIN communication according to the chosen LDF.

7 Pseudo code for minimalistic driver integration:

```
int main(void)
{
  l_sys_init(); /* initialize LIN stack */
  while(1)
  {
    l_cyclic_com_task(); /* call task every loop */
    if(1ms_timer) /* call function every 1ms */
    {
      ld_task();
    }
  }
}

void interrupt_handler (void)
{
  if (RX IRQ)
  {
    l_ifc_rx(); /* call ISR function */
  }
}
```

8 Other Recommended Calls

8.1 l_ifc_read_status()

This statusword is defined for LIN 2.0 and LIN 2.1 and contains useful state information about the LIN communication. The word is deleted once it is called, so if there is information available, the user can be sure that it is new information.

The statusbyte consist of 16 bit where the higher 8 bits contains the PID of the last transmitted frame and the lower 8 bits are status information.

8.2 Frame and Signal Update Flags

For subscribed frames there exists a update flag framework that allows checking for new available information independently from every other state information. If an update flag is set the frame was successfully received and the data is consistent and can be used by application. Additional checks according to valid range and values may be user specific and have to be implemented by the application engineer accordingly. The macro for flag checks and clearing can be found in the genlinconfig.h file (part Signal Access Macros).

8.3 Read and Write Signal Access

These macros are as defined by LIN specifications. It allows easy access to LIN signals without bothering about their actual position in the frame data or byte borders. They are also found in the genlinconfig.h file (part Signal Access Macros).

9 Diagnostic Message Handling

The driver stack supports different kinds of diagnostics. The simplest ones are the node configuration services specified in LIN specification. The LDC-Tool has a tab for configuring them separately. Each service can be activated or deactivated. No additional diagnostic API is needed for this (Class I).

Classes II and III need additional services which are not covered by the LIN specification. There are 3 different ways to achieve this.

The driver demo does not include diagnostic or node configuration services.

9.1 User defined diagnostics

This method allows the application engineer to define the behaviour of the driver upon receiving diagnostic messages on his own.
No retransmission is made on errors, so complete flow control has to be implemented in the application.

This method has the advantage that the code size of the driver is smaller than with the other two APIs. The LDC-Tool gives here the option to at least activate the mandatory node configuration services (otherwise they have to be implemented anew).

A small example is given in the main.c of the driver delivery.

9.2 Cooked API

This API has a very good flow control, and diagnostic message headers and formatting is done by the driver itself. Supports multi frame diagnostics automatically. This API does not allow setting the first three bytes of a single frame (NAD, PCI, and SID) and the first four bytes of a multiframe message (NAD, PCI, LEN, and SID). This formatting is defined by LIN specification.
If a transmission error occurred, frame is automatically retransmitted. No header is responded to if no diagnostic message was buffered.

Example:

```
/* Diagnostic Handling */
switch (lin_diag_statemachine)
{
  case 0u:
    /* Start diagnostic state machine
     * Prepare buffer and wait for masterrequest */
    ld_receive_message (&lin_diag_buffer_length, lin_diag_buffer);
    lin_diag_statemachine++;
    break;
  case 1u:
    if (ld_rx_status() > LD_COMPLETED)
    {
      /* some error occured, dummy implementation - reset transport layer
       * and restart diagnostic */
      ld_init();
      lin_diag_statemachine = 0u;
    } else {
      if (ld_rx_status() != LD_IN_PROGRESS)
      {
        /* Buffer is filled with data, check data and prepare response */
        /* dummy implementation - send data received back as response */
        ld_send_message (lin_diag_buffer_length, lin_diag_buffer);
        lin_diag_statemachine++;
      }
    }
    break;
  case 2u:
    if (ld_tx_status() > LD_COMPLETED)
    {
      /* some error occured, dummy implementation - reset transport layer
       * and restart diagnostic */
      ld_init();
      lin_diag_statemachine = 0u;
    } else {
      if (ld_tx_status() == LD_COMPLETED)
      {
        lin_diag_statemachine = 0u;
      }
    }
    break;
  default:
    break;
}
```

9.3 Raw API

This API also supports multi frame diagnostics. But assembling and disassembling of messages as well as formatting has to be done by the application engineer. This API allows setting all bytes of a diagnostic frame as by the application engineer desired. The first bytes are not restricted to NAD, PCI and SID information.
If a transmission error occurred, frame is automatically retransmitted. No header is responded to if no diagnostic message was put into queue.

Example:

```
/* Diagnostic Handling */
switch (lin_diag_statemachine)
{
  case 0u:
    /* Start diagnostic state machine
     * Prepare buffer and wait for masterrequest */
    if (ld_raw_rx_status() == LD_DATA_AVAILABLE)
    {
      ld_get_raw(data_buffer);
      lin_diag_statemachine = 1;
    }
    break;
  case 1u:
    if (ld_raw_tx_status() <= LD_QUEUE_AVAILABLE)
    {
      /* put two frames into transmit queue*/
      ld_put_raw (data_buffer);
      lin_diag_statemachine++;
    }
    break;
  case 2u:
    if (ld_raw_tx_status() > LD_TRANSMIT_ERROR)
    {
      /* some error occured, dummy implementation - reset transport layer
       * and restart diagnostic */
      ld_init();
      lin_diag_statemachine = 0u;
    } else {
      if (ld_raw_tx_status() == LD_QUEUE_EMPTY)
      {
        lin_diag_statemachine = 0u;
      }
    }
    break;
  default:
    break;
}
```