

Using NXP’s LIN Driver with the MagniV Family

by: Manuel Rodríguez
Agustin Diaz

1 Introduction

The S12 MagniV family targets the design of the smallest LIN nodes for both automotive and industrial applications providing a highly integrated and highly reliable solution. In order to decrease the development time of applications NXP provides a LIN 2.x/SAE J2602 compliant driver for its devices.

This document is intended to provide an introduction to NXP’s LIN driver, the target audience is expected to have some experience with the LIN communication protocol, for a deeper treatise of the LIN communication protocol or the driver please refer to the LIN specification document or the LIN driver manual. This application note will rely on a “hands-on” example to get the LIN driver up and running. Moreover, the application note will provide some guidelines about using some functionalities like different frame types, configuring slaves through LIN and considerations for going to sleep and waking up through LIN.

Contents

1	Introduction	1
2	MagniV family overview.....	2
3	Example overview	2
4	Node Configuration Tool.....	3
5	System initialization	11
6	API overall description.....	14
7	Frame types.....	15
8	Configuring slaves through LIN.....	28
9	Go to sleep command and wake up request	31
10	References.....	34



2 MagniV family overview

MagniV's increasing portfolio of highly integrated solutions features several microcontrollers aimed to allow the design of the smallest LIN nodes. Each of these microcontrollers is built in such a way that it spans the majority of applications that might require a LIN interface, such as window lifters, sunroof controllers, wipers, body and control solutions. MagniV LIN portfolio has been created to fulfill the requirements of the automotive industry and work in space constrained environments such as those encountered in some automotive applications.

Figure 1 shows MagniV's LIN portfolio and some of its target applications.

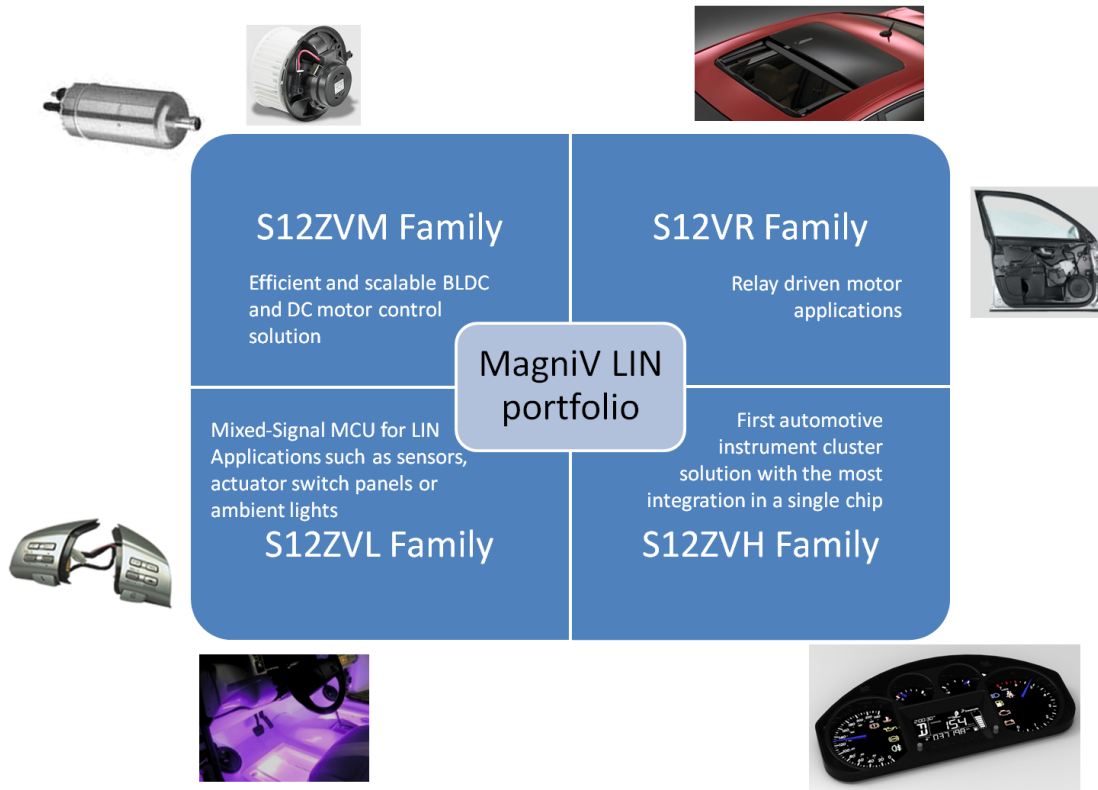


Figure 1- MagniV Portfolio

Figure 1 features MagniV's LIN portfolio to the date of the document, our MagniV portfolio is constantly increasing please visit www.nxp.com/magniv for the updated portfolio.

3 Example overview

In this document a simple LIN application is developed, a [S12ZVM Evaluation Board](#) is being used as master and a [S12VR64 Evaluation Board](#) is used as slave.

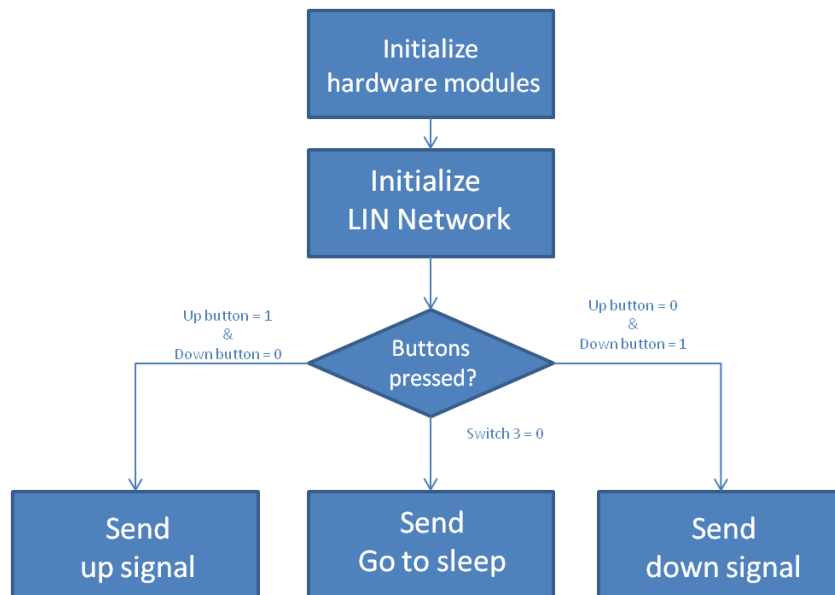


Figure 2 - Application flow diagram

Figure 2 features an overview of the application flow diagram. The S12ZVM board reads the state of the buttons and based on them sends the corresponding signals to the S12VR64 board to activate a relay, thus simulating the control of a DC motor. In the following sections it will be shown how to initialize the LIN Network using NXP’s LIN driver and how to setup MagniV hardware to work in conjunction with NXP’s LIN driver.

This application note also contains different examples of the other mentioned topics. In each of the sections it is described what is being done in the example and illustrations regarding its functionalities are shown. All this examples were developed using [S12ZVL32 Evaluation board](#).

4 Node Configuration Tool

The node configuration tool allows the user to generate in a straightforward manner the c files that describe the network architecture, signals and schedule for the network to follow. The node configuration tool takes as inputs a LIN Description File (LDF) and a Node Private File (NPF) and outputs the necessary c files for a node to integrate into the network. The LDF file describes the network architecture (master and slaves), the signals that each node “publishes” and “subscribes to”, the schedule tables for the network, speed of the network (baud rate) and version of the protocol to be used. The NPF file contains the node name, communication channel in use (e.g. SCI, UART, SLIC), clock frequency and port used. Every LIN cluster requires one LDF file and one NPF file per node in the network.

The workflow of the node configuration tool is being depicted in Figure 3. It can be seen how the c files generated by this tool, in combination with the LIN driver, once compiled yield the required files in a LIN network for each of the supported devices.

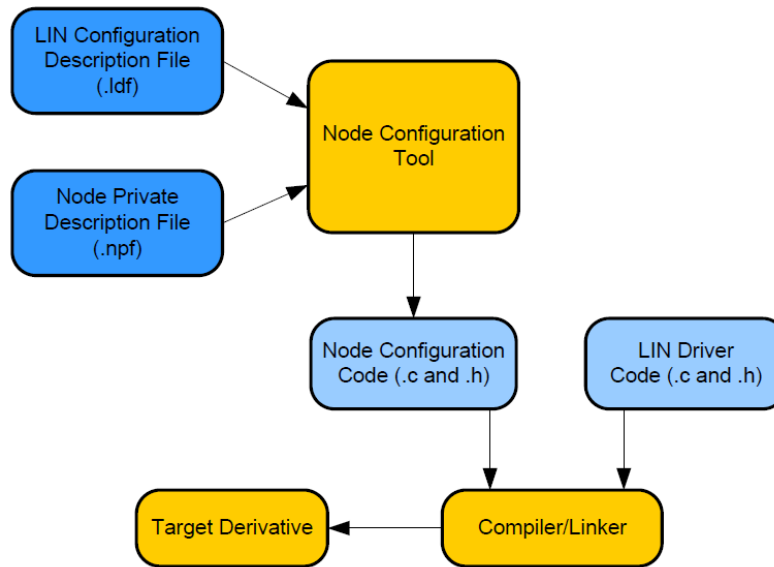


Figure 3- Node Configuration Tool workflow

4.1 LDF file and NPF file creation

The node configuration code can be generated in three ways:

- Windows command line
- Standalone Graphical User Interface (GUI)
- Eclipse plug-in

The first two methods require the creation of the LDF and NPFs files by hand; in contrast the last option includes an intuitive GUI that aids in the creation of the LDF and NPFs files. The Eclipse plug-in option has been chosen in this document since it significantly decreases the development time. For details of the remaining methods please refer to the corresponding user guide in the NXP LIN driver package.

Please follow the instructions in the Eclipse user manual chapter 3.1 Plug-in Setup, contained in the documentation folder of the NXP LIN driver package. A project must be created in order to work with this tool, please open CodeWarrior Development Studio and click File>New>Project... as in Figure 4.

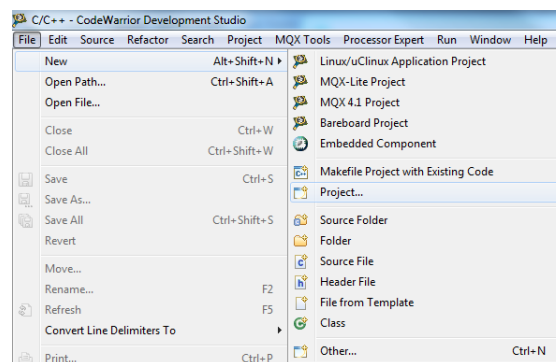


Figure 4-Creating a new project in CodeWarrior

Click on “General” and select “Project” as in Figure 5.

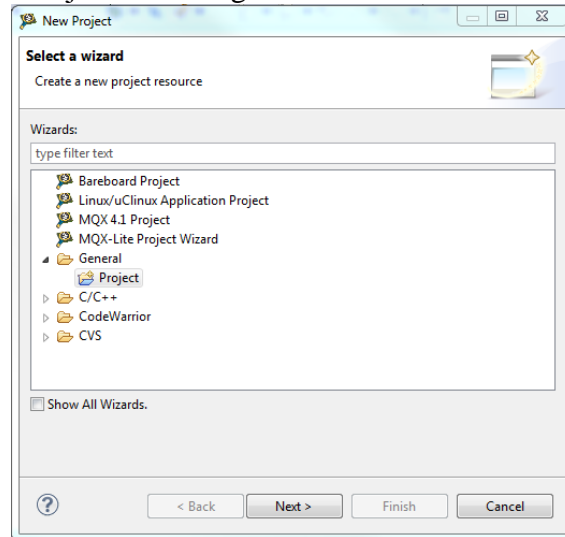


Figure 5- Creating a new project in CodeWarrior

Click “Next>” select the name of the project and click “Finish”.
Right click the project Folder, select “New>Other” as in Figure 6.

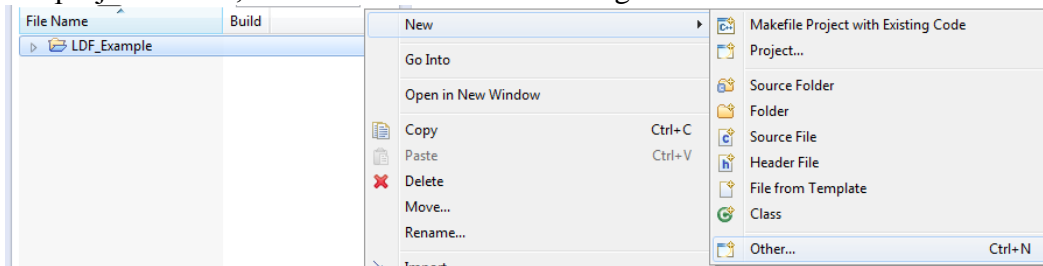


Figure 6- Creating LDF and NPFs files

Click on “LIN Node Configuration Files”, select “LDF file” name the file and click “Finish”. Figure 7 for reference.

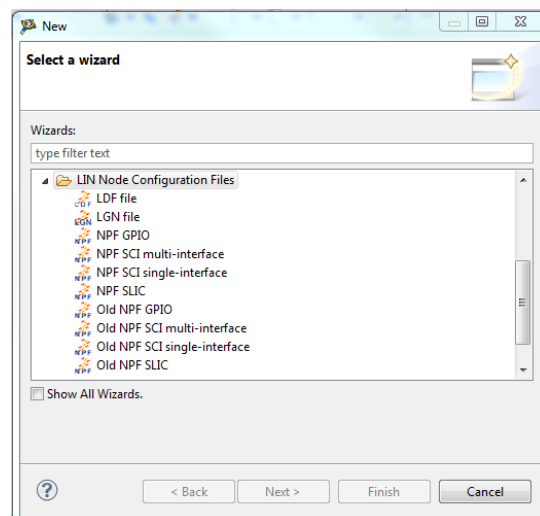


Figure 7- Creating LDF and NPFs files

Follow the procedure for the creation of a LDF file but select “NPF SCI single-interface”, this file must be chosen since MagniV devices have the LIN physical layer routed to a SCI channel, name the file and click “Finish”. Two different instances of this file must be created, one for the master device and one for slave (a NPF file must be created for each node in the network). If another device is intended to join the network the corresponding NPF file must be selected with respect to the device peripherals.

4.2 LDF file configuration

Double click on the LDF file that has been created to open it, then select the GUI editor as in Figure 8. A warning will appear and the GUI editor will be shown.

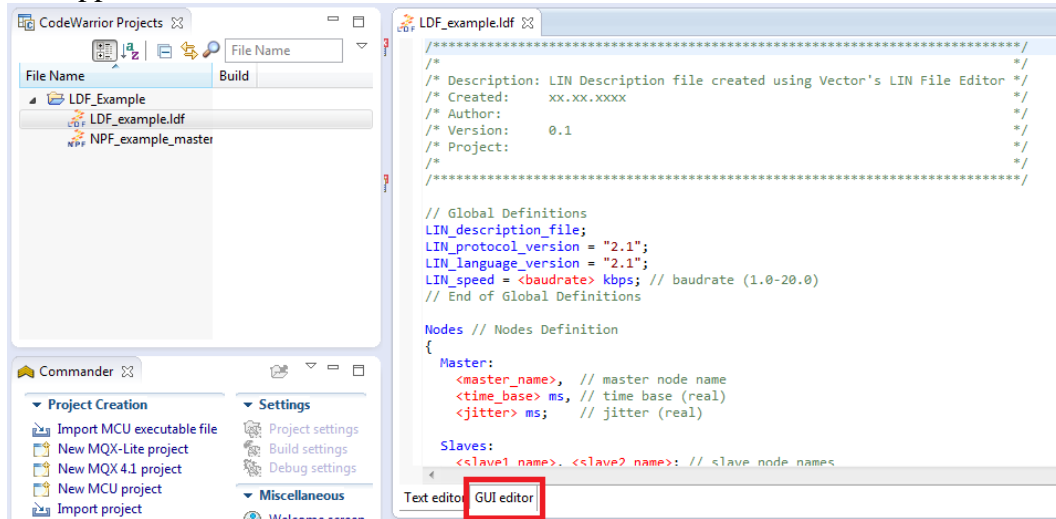


Figure 8- Configuring the LDF file

Select the protocol version and speed for the application as in Figure 9.

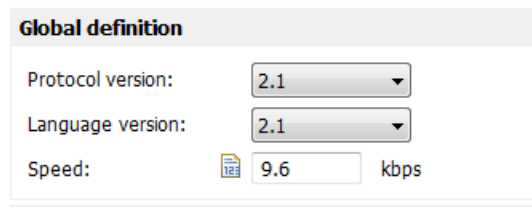



Figure 9-Configuring the LDF file

In the “Node definition” section name the master device, set the time base (usually between 5ms and 10ms, LIN specification for details) and jitter. Click on the  button to add a slave node, a pop-up window will appear prompting for the name of the slave. See Figure 10 for reference.

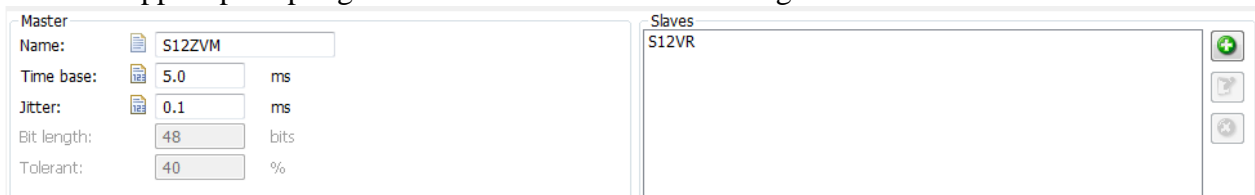



Figure 10-Configuring the LDF file

Move on to the “Signal definition” section, in this example three signals are being defined:

- status – Holds the status of the “window”.
- up_down – Holds the command for the slave node controlling the “window”.
- error – Error signal used required by the slave.

To add these signals the  button must be pressed, a pop-up window will appear and it must be configured as in Figure 11 with respect the required signal properties.

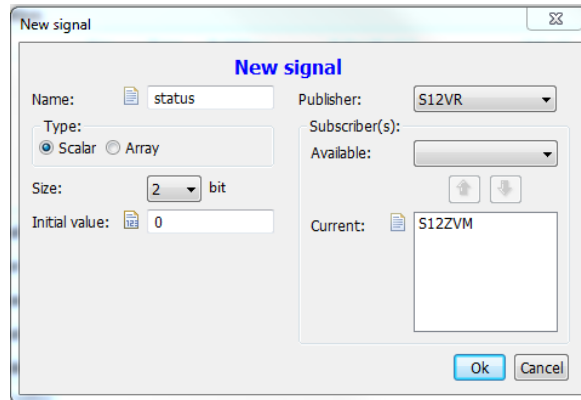



Figure 11- Signal configuration

The defined signals for this example can be seen in Figure 12.

Signal definition					
Name	Size ...	Type	Publisher	Subscriber(s)	Init val...
status	2	Scalar	S12VR	S12ZVM	0
up_down	2	Scalar	S12ZVM	S12VR	0
error	1	Scalar	S12VR	S12ZVM	0

Figure 12-Signals

In this example only unconditional frames are being used, nevertheless the GUI is capable of generating all the frame types specified by the LIN consortium, i.e. Unconditional Frame, Event triggered frames, sporadic frames and diagnostic frames. As for the signal definition, to define an unconditional frame the  button must be pressed and the following window must be configured. See Figure 13 as a reference.

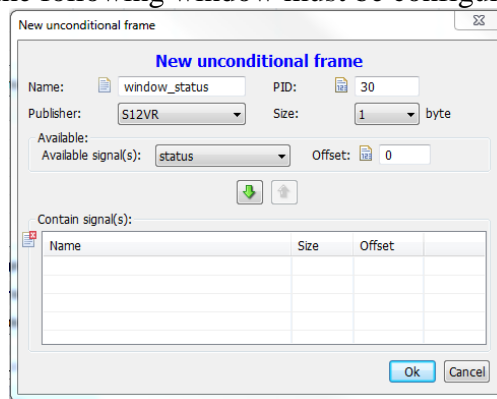


Figure 13- Unconditional frame definition


In this example two unconditional frames are defined, one for the status of the “window” and another one for the up and down command. PIDs are being chosen randomly.

Unconditional frame definition				
Name	ID	Published by	Size ...	Signal(s)
window_status	30	S12VR	1	status,error
up_down_command	13	S12ZVM	1	up_down

Figure 14-Unconditional frames definition

As can be seen in Figure 14 the “S12VR” node has two signals associated with its frame, these signals must be added with an offset so that they do not overlap, e.g. status offset 0 (size of 2 bits), error offset of 2 bits to place it after the “status” signal.

Since the other frames will remain unused in this example, these can be left unchanged. The same procedure as in the unconditional frame could be followed in order to configure the remaining frames if required.

To configure the “Node attribute definition” section, the  button must be pressed, and the pop-up window must be set as in Figure 15.

Frame name	PID	Message ID
window_status	30	
up_down_command	13	

Figure 15-Node attribute configuration

The configured Node Address (NAD) and initial NAD have been chosen randomly as is up to the designer to choose its value. The configured NAD specifies the diagnostic address; this is used in the automatic conflict resolving procedure, as well as in diagnostic and configuration functions, to identify the node. For this reason, it is a unique number within the cluster.

The product ID is a mandatory part of the node configuration; it consists of the supplier ID¹, the function ID², and the variant ID³.

The response error identifies the signal name used for LIN error reporting, and must be defined in the signal section.


The time value P2 min defines the minimum time for a slave to prepare its response to a master request frame and its default value is 50ms (refer to the LIN specification package for details).

The ST min time defines the minimum time between two slave response frames and its default value is 0ms.

The N_As_timeout defines the maximum time for transmission of a master request frame on the transmitter side and its default value is 1000ms.

The N_Cr_timeout defines the maximum time until reception of the next consecutive frame and its default value is 1000ms (refer to the LIN specification package for details chapter 3.2.5 Timing constraints).

Configurable frames must list all frames (unconditional frames, event triggered frames and sporadic frames) processed by the slave node.

To move onto the next and last step of the LDF file configuration the “Schedule table definition” must be selected and the  button must be pressed. The window in Figure 16 will pop-up.

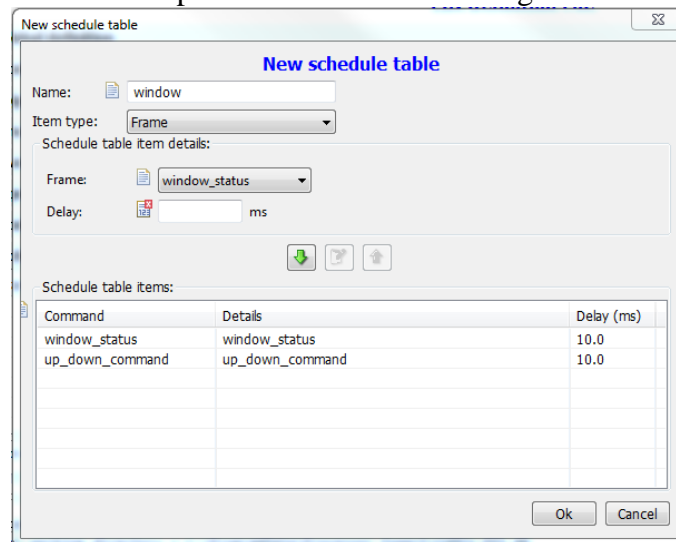



Figure 16- Schedule table configuration

Name the schedule table and select the type of items it will hold. In this example the only schedule table that will be used is intended to poll the status of the “window” and send the “up and down command” therefore the item type has been selected as “frame”. The frames related to this schedule table must be added and a delay between them must be defined, the delay specified for every schedule entry shall be longer than the jitter and the worst-case frame transfer time.

The file must be verified by pressing  button in your CodeWarrior environment and a window must pop-up with the following message “This LDF form is correct”.

¹ The supplier ID is assigned by the LIN consortium. For NXP it is 0x000B (11).

² The function ID is assigned by the supplier; in this case 0x0020 (18) was chosen.

³ The variant ID must be changed whenever the product is changed, but with an unaltered function.

4.3 NPF file configuration

Double click on the NPF file for the node that will be configured to open it, then select the GUI editor. A warning will appear and the GUI editor will be shown. Select the “Global definition” section to configure it.

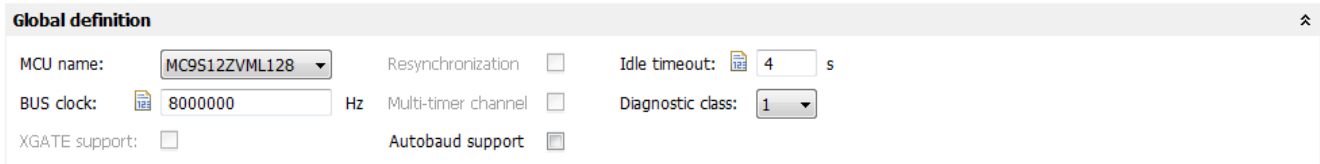



Figure 17- NPF file configuration

Select the device that corresponds with the node, in this case the master node is an S12ZVML128, with a bus clock frequency of 8MHz. The idle timeout will send the bus to sleep after this time has elapsed the minimum time for this is 4s and the maximum is 10s as per the LIN specification. Autobaud support must not be selected in the master node.

In this case diagnostic class 1 has been chosen for the application. The diagnostic class must be selected with respect to the node function. For more information regarding diagnostic classes go to section 7.4 *Diagnostic Frame*.

Onto the next section “Hardware definition”, the  button must be pressed and the window that pops-up must be configured.

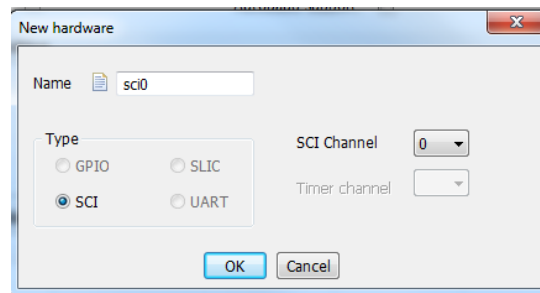


Figure 18-Node hardware configuration

Name the “new hardware” and select the SCI channel that is routed to the LIN physical layer of your MagniV device, in this case SCI channel 0 is routed to the LIN physical layer of both S12ZVM and S12VR devices. The new configured hardware is shown in Figure 18.

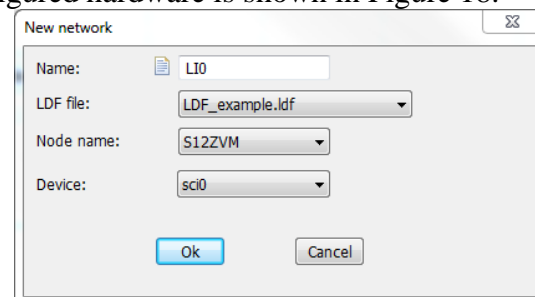




Figure 19-Network configuration

Name the “new network”, select the corresponding LDF file, name of the node and the previously defined hardware device and click “Ok”. The new configured network is shown in Figure 19.

The file must be verified by pressing  button in your CodeWarrior environment and a window must pop-up with the following message “This NPF form is correct”.

To generate the c files for this node click on . A window will prompt for the output folder select it and click “OK”. Figure 20 shows the selected folder for this application.

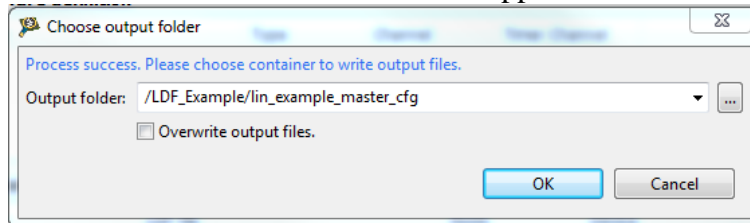


Figure 20-NCF generate files

A new folder must be created and the following text must appear in the console “Generate completed. Output files already written in folder: /Selected_folder”.

The same procedure must be followed for the slave node, but in this case the Autobaud support can be selected if desired.

The slave configuration used in this example is being showed in Figure 21.

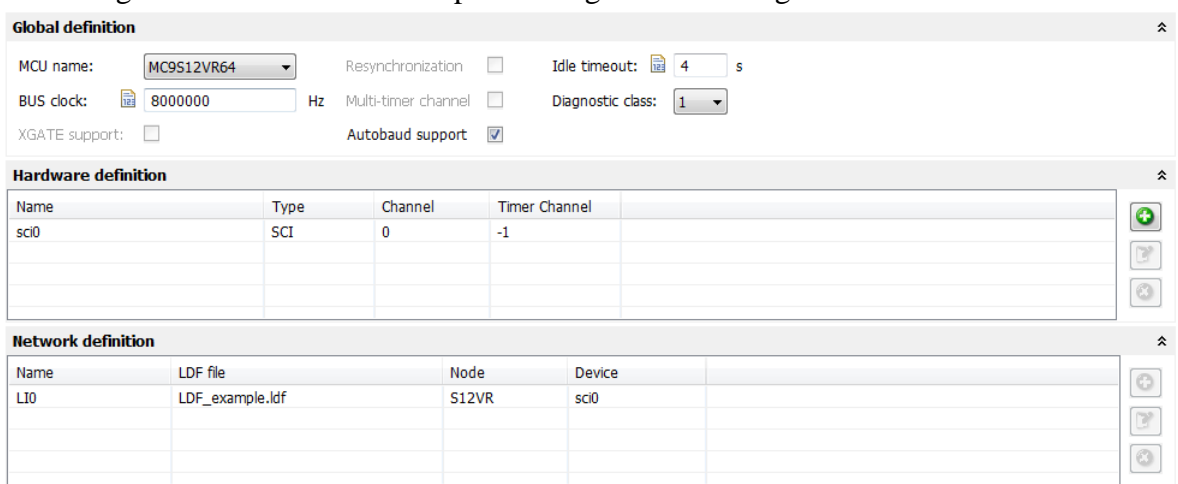




Figure 21- Slave configuration

To generate the c files for this node click on . A window will prompt for the output folder select it and click “OK”. A new folder must be created and the following text must appear in the console “Generate completed. Output files already written in folder: /Selected_folder”.

5 System initialization

A new project must be created for each device in the network and the LIN driver sources must be dragged and dropped into each project. For S12Z (CodeWarrior 10.6) microcontrollers the paths must be added, Right click on the project->properties->C/C++ Build->Settings->S12Z Compiler-> Access Paths

and click on the  button in the “Search User Paths” section. Click on Workspace... select all the folders of the LIN driver and click “ok”. See Figure 22 and Figure 23 for reference.

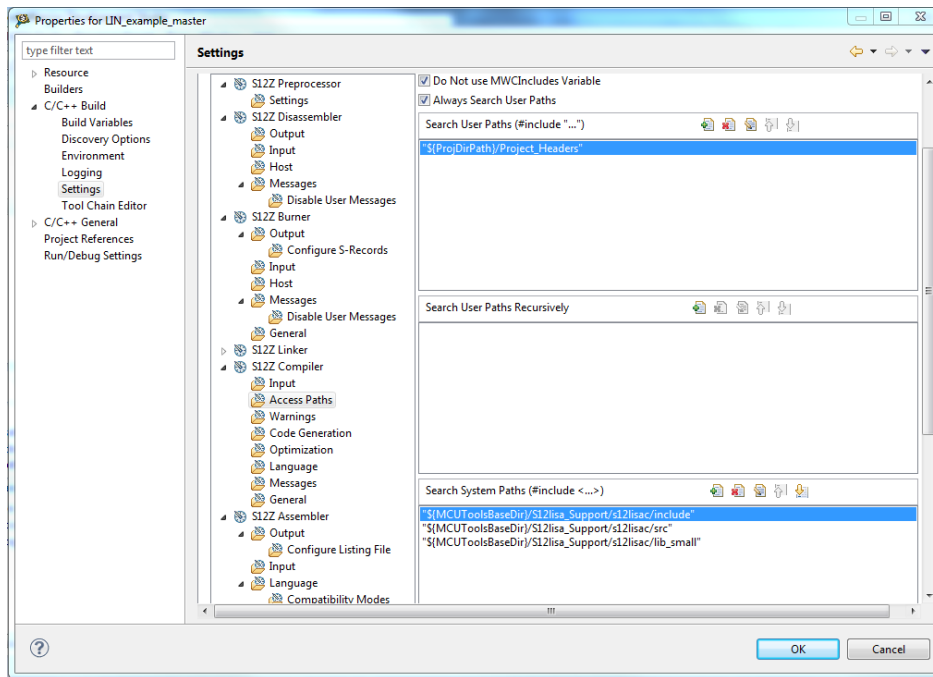


Figure 22-Updating access paths

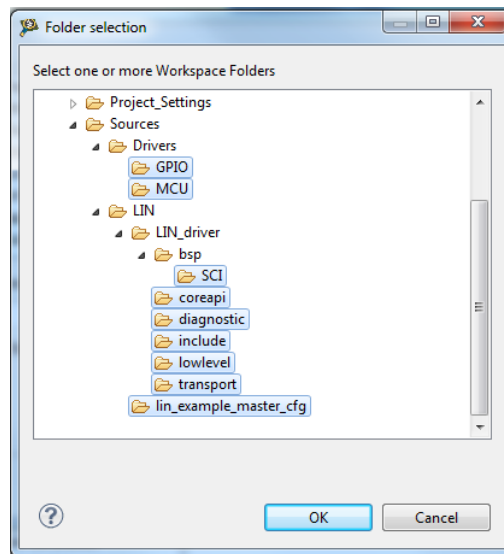


Figure 23-Add access paths

For S12 microcontrollers (CodeWarrior Classic IDE) simply drag and drop, the access paths will be updated automatically. To be able to use the LIN driver in your project include the header definition “lin.h” in your document.

The hardware must be configured accordingly with the NPFs files, therefore the clock frequency of each microcontroller has to be adjusted, please refer to the reference manual of your device to perform the required clock adjustments.

The LIN physical layer of the MagniV device must be enabled, the name of the registers might vary from device to device, but the procedure remains the same at a high level. The following pseudo-code illustrates the general procedure to enable the LIN physical layer of a MagniV device, please refer to the device reference manual.

```
SELECT_PULL_UP;    // This is done to strengthen the signal
SELECT_SLEW_RATE;  // Optimizes the behavior for the selected baudrate
ENABLE_PHYSICAL_LAYER; // Enable LIN physical layer
```

The LIN network must be initialized before trying to initiate communication. This is performed by calling the following routines declared in the LIN specification package.

- `l_sys_init(void)`: Performs the initialization of the LIN core (timer initialization). The call to the `l_sys_init` is the first call a user must use in the LIN core before using any other API functions.
- `l_ifc_init(l_ifc_handle iii)`: Initializes the controller specified by the name `iii` (network name specified in the NPF file), i.e. sets up internal functions such as the baud rate. The default schedule set by the `l_ifc_init` call will be the `L_NULL_SCHEDULE` where no frames will be sent and received. This is the first call a user must perform, before using any other interface related LIN API functions.

For a slave node this is the end of the initialization process for the LIN network, beside this it might be necessary to initialize other modules for other applications but this relies entirely on the application therefore is not covered in this application note. The last point is to enable the interrupts, this can be done with the command “EnableInterrupts;”.

For a master node a last step is required, the schedule table to be followed must be selected and a routine must be initialized to follow such schedule, a timer set as “output compare” is commonly used for this purpose.

The following routines must be called:

- `l_sch_set(l_ifc_handle iii,`
 `l_schedule_handle schedule_iii`
 `l_u8 entry)`

`iii` is the interface name (network name specified in the NPF file), `schedule_iii` is the name of the schedule table to be followed, `entry` is the entry point for the schedule table (the frame to be transmitted at the next “tick”).

This routine sets up the next schedule to be followed by the `l_sch_tick` function for a certain interface. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point. This routine must be called after “`l_ifc_init`”

- `l_sch_tick(l_ifc_handle iii)` This function follows a schedule. When a frame becomes due, its transmission is initiated. When the end of the current schedule is reached, this function starts again at the beginning of the schedule. This function must be called periodically by the master every “`time_base`”, this is normally called at the ISR of the configured timer.

Care must be taken not to use the same timer that the driver uses for the schedule. Please see Table 1 for reference, or refer to the LIN user manual in the LIN driver package for an updated table.

Table 1-Channels used

MCU	Timer	Channel used
S12ZVM128	Tim0	0
S12ZVL32		
S12ZVHY64		
S12VR64		

Once this has been configured the master is ready to be used and the interrupts must be enabled.

6 API overall description

The LIN API is a network software layer that hides the details of a LIN network configuration (e.g. how signals are mapped into certain frames) for a user making an application program for an arbitrary ECU. NXP's LIN driver integrates this API and is completely compliant with the LIN specification package. This application note only covers functions related with:

- Reading and writing a signal.
- Send “Go to sleep” command and “wake up” command

For a detailed explanation of the available functions please refer to the LIN specification package or the FSL_LIN_API_Documentation webpage included in the LIN driver package (“FSL_LIN_DRIVER”/Documentation/ FSL_LIN_API_Documentation).

6.1 Reading and writing a signal

The available signal types are the following:

- l_bool For one bit signals; zero if false, non-zero otherwise
- l_u8 For signals of the size 2 - 8 bits
- l_u16 For signals of the size 9-16 bits

To read a scalar signal one of the following functions must be used, depending on the type of the signal to be read:

- l_bool l_bool_rd_sss(void);
- l_u8 l_u8_rd_sss(void);
- l_u16 l_u16_rd_sss(void);

Where “sss” is the signal handle which is the name of the network underscore name of the signal, therefore if the previously defined signal status must be read, this can be achieved by calling the function “l_u8_rd_LI0_status()”

To read an array the following function must be used:

- l_bytes_rd_sss (l_u8 start, /* first byte to read from */
l_u8 count, /* number of bytes to read */

```
l_u8* const data);          /* where data will be written */
```

This function reads and returns the current values of the selected bytes in the signal. The sum of start and count shall never be greater than the length of the byte array.

To write a scalar signal one of the following functions must be used, depending on the type of the signal to be written:

- `l_bool` `l_bool_wr_sss(l_bool v);`
- `l_u8` `l_u8_wr_sss(l_u8 v);`
- `l_u16` `l_u16_wr_sss(l_u16 v);`

Where `v` is the value to be written to the signal and “`sss`” is the signal handle as has been already stated.

To write an array the following function must be used:

- `l_bytes_wr_sss (l_u8 start, /* first byte to write to */
l_u8 count, /* number of bytes to write */
const l_u8* const data); /* where data is read from */`

Sets the current value of the selected bytes in the signal specified by the name `sss` to the value specified. The sum of start and count shall never be greater than the length of the byte array, although the device driver may choose not to enforce this in runtime.

7 Frame types

Each of the frames types described below have their own code examples.

7.1 Unconditional frame

This is the standard LIN Frame type. It is used for cycle communication between LIN nodes. The frame identifiers for this kind of frames goes from 0 to 59 (0x3B). The header of an unconditional frame is always transmitted and it is associated with only one publisher, which can be either master or slave, and can be associated with multiple subscribers. The publisher should always provide a response to the unconditional frame header and the response can be either received by a single subscriber node or multiple nodes.

The example provided by the section above provides the procedure to set up unconditional frames.

7.2 Event triggered frame

This frame type was introduced in LIN 2.x. The frame identifiers for this kind of frames goes from 0 to 59 (0x3B). The purpose of an event triggered frame is to reduce the bus bandwidth assigned to poll multiple slave nodes looking for a change in the associated signal. Every event triggered frame has one or several additional frames associated to it. The main difference is that subscribers related to an event triggered frame only respond if their data have changed instead of every time the header is transmitted.

All the subscriber of the event triggered frame shall receive the frame, process its data and respond if any of the signals associated with that frame have changed.

The associated unconditional frames to an event triggered frame must meet the following:

- Must have the same length
- Must have the same checksum model
- Must have no signal in the first data byte

The structure of an event triggered frame is the following:

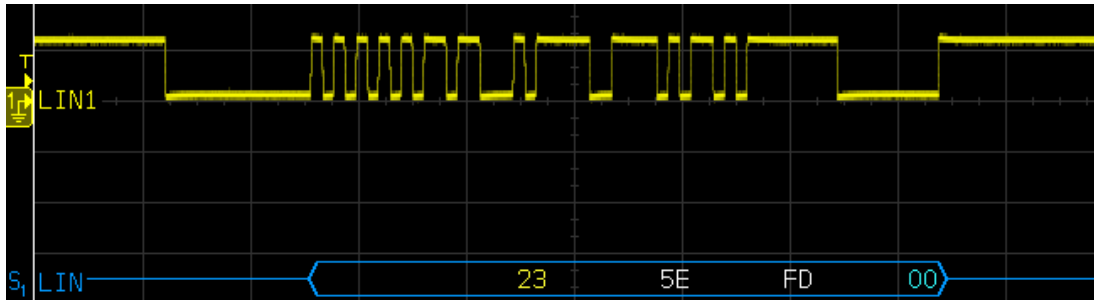


Figure 24-Add access paths

- 1) Break field plus syncfield
- 2) Event triggered frame ID (in this example 0x23)
- 3) First byte of data field must be the ID of the related subscriber's unconditional frame which data has changed (in this example 0x5E).
- 4) Data of the unconditional frame (in this example 0xFD).
- 5) Checksum (in this example 0x00).

If two or more subscribers of the event triggered frame try to respond at the same time a collision will occur. In this case the master should have a collision table defined in order to poll each of the devices in order to find which of the slaves tried to respond.

The following figures will illustrate how to set up an event triggered frame using the NXP's LIN driver. The procedure will not focus on setting up a network from nothing but the special considerations that must be taken in order to set up an event triggered frame.

In this example there are two available signals published by the slave. One is the error signal and the other one is the status of a button. Both signals are 1 bit in size.

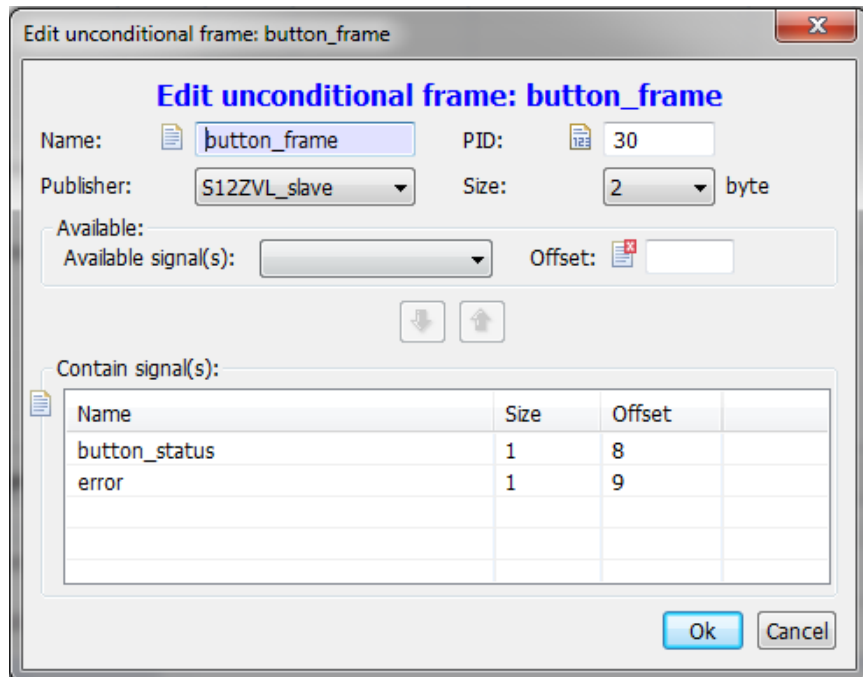


Figure 25. Unconditional frame related to event triggered frame

First of all the unconditional frame related to the event triggered frame must be defined. It is important to consider that this unconditional frame must have the first by reserved for the ID. In this example both signals are placed in the second byte of the frame setting the offset to 8 and 9 bits respectively. The next step is to set up a collision table:

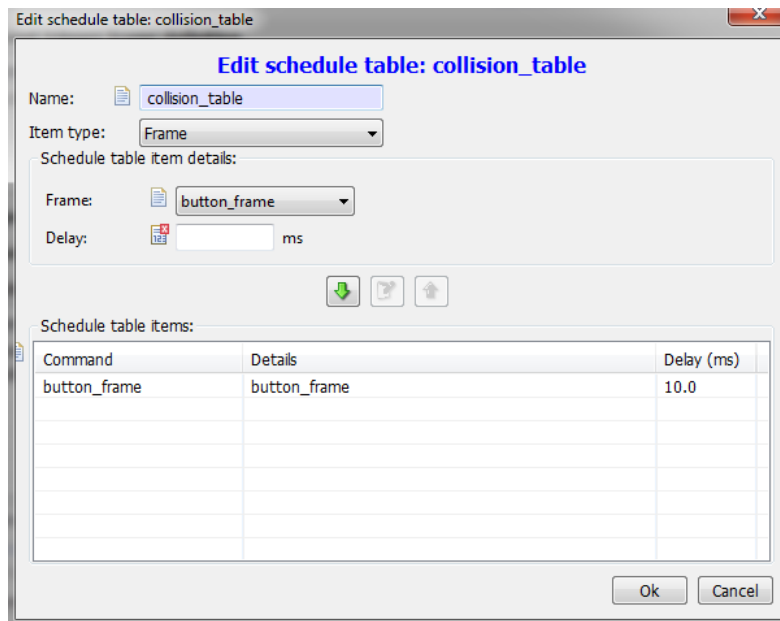


Figure 26. Collision schedule table definition

The collision table will be activated when a collision occur in the event triggered frame. In this schedules each of the slave nodes should be polled in order to find out which tried to answer. For this example only one slave node is defined. Next step is to define the event triggered frame:

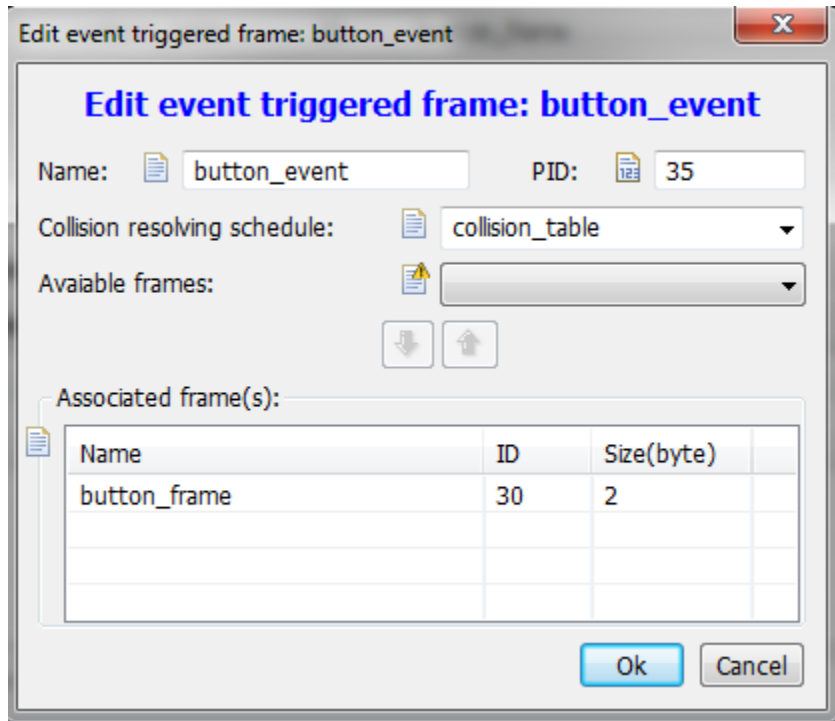


Figure 27. Event triggered frame definition

An ID between 0 and 59 must be chosen for the frame. Then the previously defined collision table must be chosen and finally each of the available frames. If an unconditional frame does not match the requirements for the event triggered frame it would not appear in the available frames.

Finally the frame must be added to the available frames of the node definition and the event triggered frame must be added to the main schedule (not the one defined for collision).

In the given example for event triggered frame there are two S12ZVL connected. One configured as master and the other as slave. The master main schedule is configured to continuously send the event triggered frame header. If the SW6 button is pressed on the slave board the button status signal change its value to 1. When this happened the slave starts to respond to the event triggered header as the following images show.

Before clicking the button:

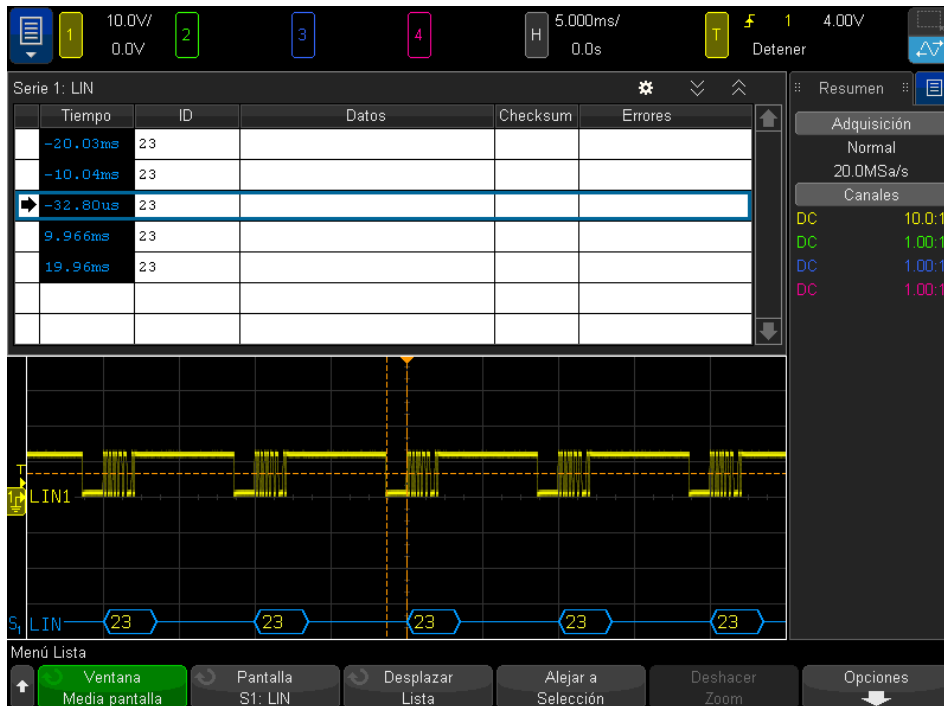


Figure 28. Event triggered frame header with no slave response

As the button has not been clicked yet the slave does not publish any response to the event triggered header.

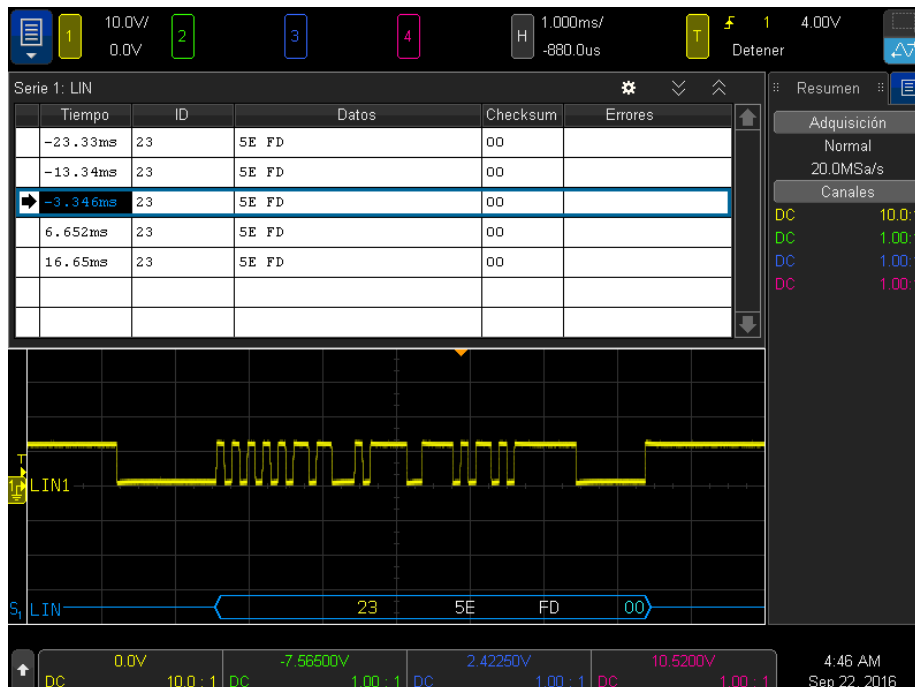


Figure 29. Event triggered frame header with slave response

Now the button has been clicked and the slave publishes its response. The first byte belongs to the unconditional frame ID (value of 30 without parity) and the rest to the button status signal, which has changed, and the error signal.

7.3 Sporadic frame

This frame type was introduced in LIN 2.x. Unlike other frame types the sporadic frame does not have an ID of its own. According to the LIN specification “The purpose of sporadic frames is to blend some dynamic behavior into the deterministic and real-time focused schedule table without losing the determinism in the rest of the schedule table.” (2010)

The sporadic frame acts as a slot holder for one or more unconditional frames. The master only sends one of these frames if its data has been updated (one or more signals have changed). If one or more signals were updated in different frames, only one of them will be sent. The one with the highest priority. However, at the next tick of the scheduler table the pending frame will be sent. If no frame will be sent (no signal has changed) the slot will remain empty and the bus silent.

When the unconditional frame has been successfully transmitted it will no longer be pending. Therefore, it would not be sent again unless any of its signals change again. The master is the only publisher of the unconditional frames related to the sporadic frame in order to guarantee that the node knows when the frame is pending for transmission.

In this example there are two available signals published by the master. One named as high priority signal and the other one low priority signal. Both are contained in a sporadic frame. When a button is pressed by the master both, the low priority signal and high priority are updated. The high priority is sent first due to the nature of the sporadic frame, then the low priority and when there are no pending frames the bus stays silent.

The following figures will illustrate how to set up a sporadic frame using the NXP's LIN driver. The procedure will not focus on setting up a network from nothing but the special considerations that must be taken in order to set up an event triggered frame.

First of all the unconditional frames related to the sporadic frames need to be set.

Name	ID	Published by	Size ...	Signal(s)
high_priority_frame	30	S12ZVL_master	1	high_priority_signal
low_priority_frame	31	S12ZVL_master	1	low_priority_signal

Figure 30. Priority unconditional frames

It is important that both frames are published by the master node.

The next step is to configure the sporadic frame.

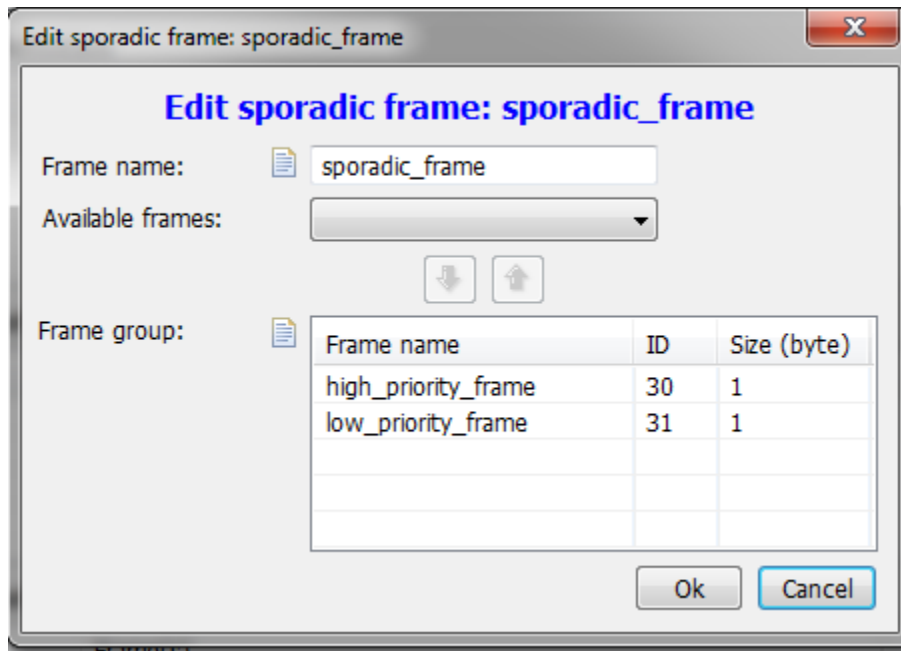


Figure 31. Sporadic frame definition

Unlike event triggered frames. The unconditional frames contained in a sporadic frame does not need any specific format. The only consideration when setting it up is that the priority will be related to the order that each unconditional frame is added to the frame group. In other words, the first on the list will be the one with the highest priority and so on.

The next step is to assign the sporadic frame to the slave node. There is a problem with the graphic tool so it must be done using the text editor.

```

/* -----NODE ATTRIBUTE DEFINITIONS----- */
Node_attributes {
  S12ZVL_slave {
    LIN_protocol           = "2.1";           /* Node protocol version */
    configured_NAD         = 0x14;           /* configured NAD of node (1-125) */
    initial_NAD            = 0x14;           /* initial NAD of node (1-125) */
    product_id             = 0x12, 0xa, 0x1; /* Product id */
    response_error         = error0;        /* Response error signal */
    P2_min                 = 50 ms;         /* P2_min */
    ST_min                 = 0 ms;          /* ST_min */
    N_As_timeout           = 1000 ms;       /* N_As timeout value */
    N_Cr_timeout           = 1000 ms;       /* N_Cr timeout value */
    configurable_frames {
      high_priority_frame;
      low_priority_frame;
      sporadic_frame;
    }
  }
}

```

Figure 32. Assigning sporadic frame to slave node

Change to the text editor using the “text editor tab.” Find the node attributes definitions and add the sporadic frame to *configurable_frames*.

Finally add the sporadic frame to the schedule table.

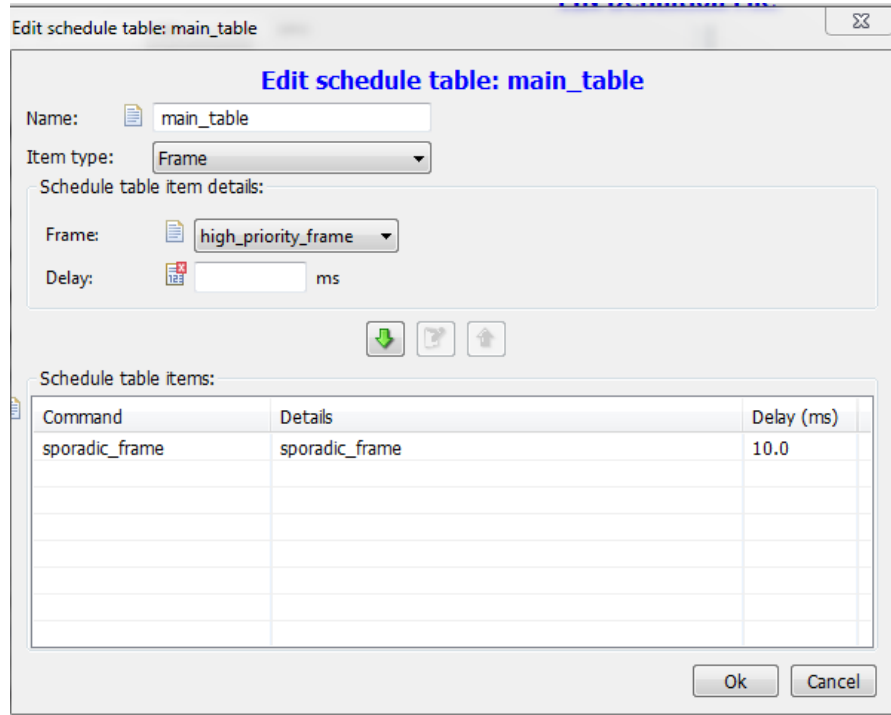


Figure 33. Schedule table for sporadic frame

In the sporadic frame example provided with this application note two signals are defined one with high priority and the other one with low priority. Each signal is contained in different unconditional frames. When a button is pressed both signals are changed simultaneously. However, due to the nature of the sporadic frame only one frame can be send at a time. The high priority will be send first and at next tick of the schedule table the following pending frame is send.

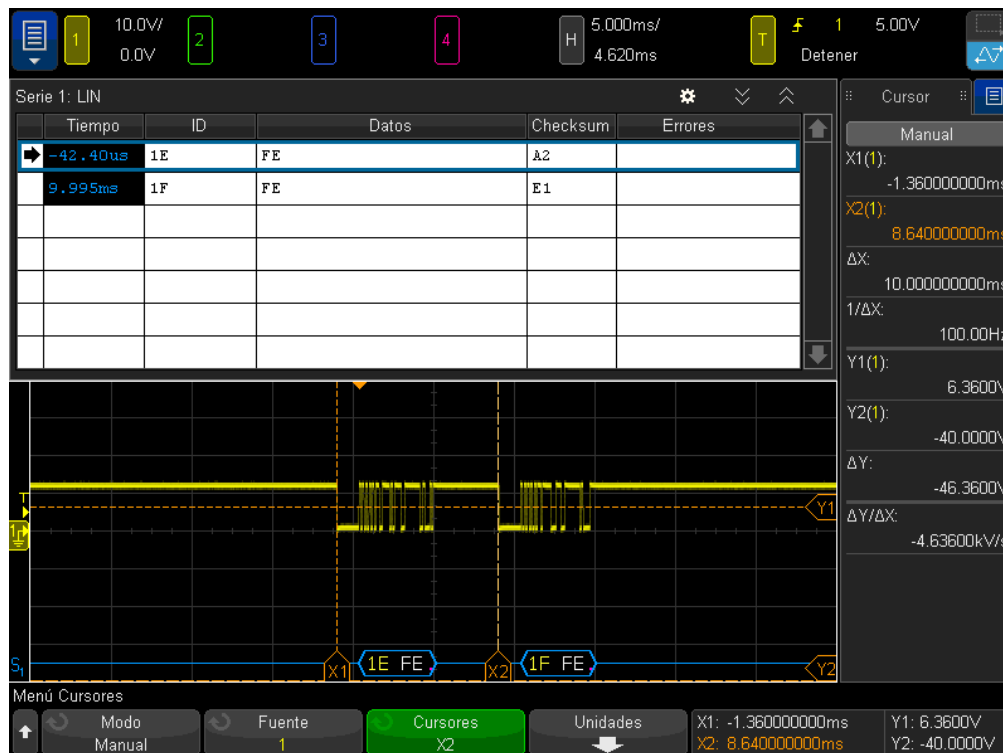


Figure 34. Sporadic frame transmission

As expected the bus was silent until any of the unconditional frames contained in the sporadic frame change. When button is pressed the high priority frame (ID = 30) is send. At next schedule tick (10 ms later) the pending low priority frame is send (ID = 31). If there are no pending frames the bus will remain silent again.

7.4 Diagnostic frame

The frame identifiers reserved for this kind of frames are 60 and 61 (0x3C and 0x3D). The purpose for this kind of frame is usually either configuration or diagnostic of the slave nodes. Both frames have fixed length of 8 data bytes and usually contain transport layer information. The two reserved frames are usually called as Master Request Frame (ID = 0x3C) and Slave Response Frame (ID = 0x3D). The Master Request Frame is always published by the master and is used to send configuration parameters to the slave. The Slave Request is usually the frame that follows a master request. It is always published by the master and is used to respond to the previous master request. The slave can either have a positive response or a negative one.

A master request frame data field is structured as the following diagram:



Figure 35. Master request data frame

A slave response frame data field is structured as the following diagram. It may vary if it is a positive response or a negative one:



Figure 36. Positive slave response



Figure 37. Negative slave response

Where:

- NAD is the Node Address for diagnostic. Each slave node has its own address.
- PCI is the Protocol Control Information. This field defines if the data will be send in a single frame of consecutive frames. Master request and slave response are always single frame. Refer to the LIN specification to how this value is formatted.
- SID is the Service Identifier. This field is defined by the diagnostic service that is provided or requested.
- D1-D5 are data transfer. If less than five bytes of data must be send. The non-used bytes are send as 0xFF.

Depending on the services that a slave node may provide it can be configured to different diagnostic classes. Here is a brief description on each of the classes:

- **Diagnostic class 1:** Smart and simple devices like intelligent sensors and actuators, requiring none or very low amount of diagnostic functionality. Actuator control, sensor reading and fault memory handling is done by the master node, using signal carrying frames. Therefore, specific diagnostic support for these tasks is not required. Fault indication is always signal based.
- **Diagnostic class 2:** Similar to a class I slave node, but it provides node identification support. The extended node identification is normally required by vehicle manufacturers. Testers or master nodes use ISO 14229-1 diagnostic services to request the extended node identification information. Actuator control, sensor reading and fault memory handling is done by the master node, using signal carrying frames. Therefore, specific diagnostic support for these tasks is not required. Fault indication is always signal based.
- **Diagnostic class 3:** Slave nodes are devices with enhanced application functions, typically doing their own local information processing (e.g. function controllers, local sensor/actuator

loops). These slave nodes execute tasks beyond the basic sensor/actuator functionality, and therefore require extended diagnostic support. Direct actuator control and raw sensor data is often not exchanged with the master node, and therefore not included in signal carrying frames. ISO 14229-1 [4] diagnostic services for I/O control, sensor value reading and parameter configuration (beyond node configuration) are required.

The following figures will illustrate how to set up diagnostic frames using the NXP's LIN driver. The procedure will not focus on setting up a network from nothing but the special considerations that must be taken in order to set up a diagnostic frames.

As diagnostic frames are usually used for diagnostic or configuring slaves it is common to use different schedules tables. One schedule to carry out all the frames that have application information like sensor values, node status, commands, etc. The other schedule to configuration of the slave node. This configuration scheduler must be used at the initialization process. When the slave is correctly configured then switch to the normal table.

For educational purposes in this example there are two schedules tables. One defined as **normal_schedule** and the other one as **diagnostic_schedule**. By default the diagnostic_schedule is selected. However, when pressing the button the selected table change to the normal_schedule. This to simulate the change in tables when slave is already configured.

For this example an Assign NAD service followed by a slave response where configured in the diagnostic_schedule. While an unconditional frame was configured in the normal_schedule. To find more information about the diagnostic services allowable in the LIN specification go to next section. Configuring Slaves through LIN.

There are two different ways in which diagnostic frames can be send. In this example it will be shown how to set them using the graphical tool of the driver as it is easier.

One thing that must be done different when setting up a diagnostic frame is the definition of the schedule. The next figure shows the two different schedule tables.

Name	Frame(s)
normal_schedule	unconditional_frame
diagnostic_schedule	AssignNAD, SlaveResp

Figure 38. Schedule tables

Normal_schedule contains one unconditional frame. While diagnostic_schedule contains two different diagnostic services that LIN specification provides. One is AssignNAD, which is a master request, and the other one SlaveResp.

To select the diagnostic services create a new schedule and click on the Item type. Different options will be displayed just like in the following figure.

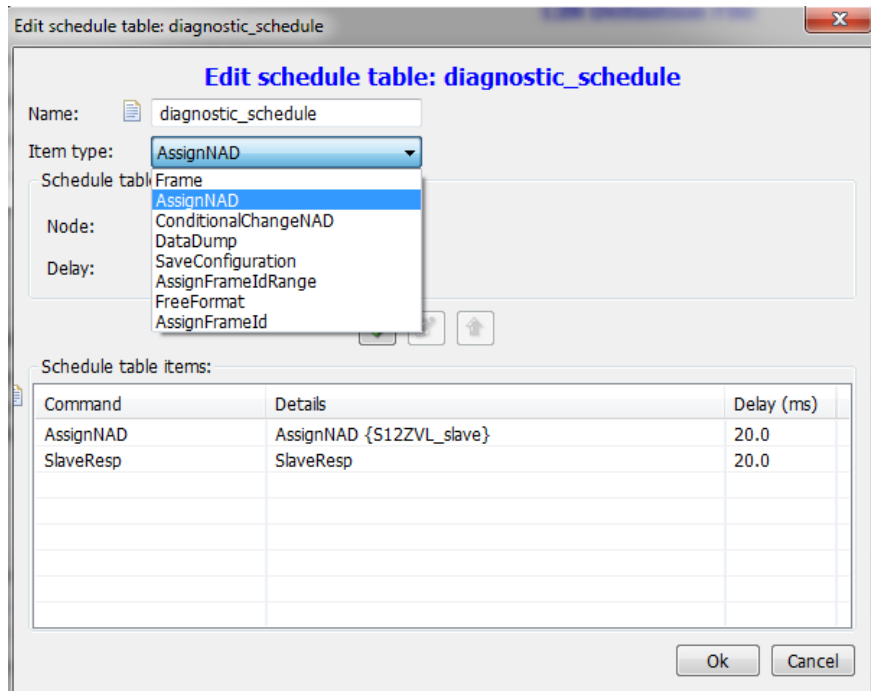


Figure 39. Selecting diagnostic services.

In this example **AssignNAD** is selected.

To select the Slave Response frame select **Frame** as Item type. At the Frame option select **SlaveResp**.

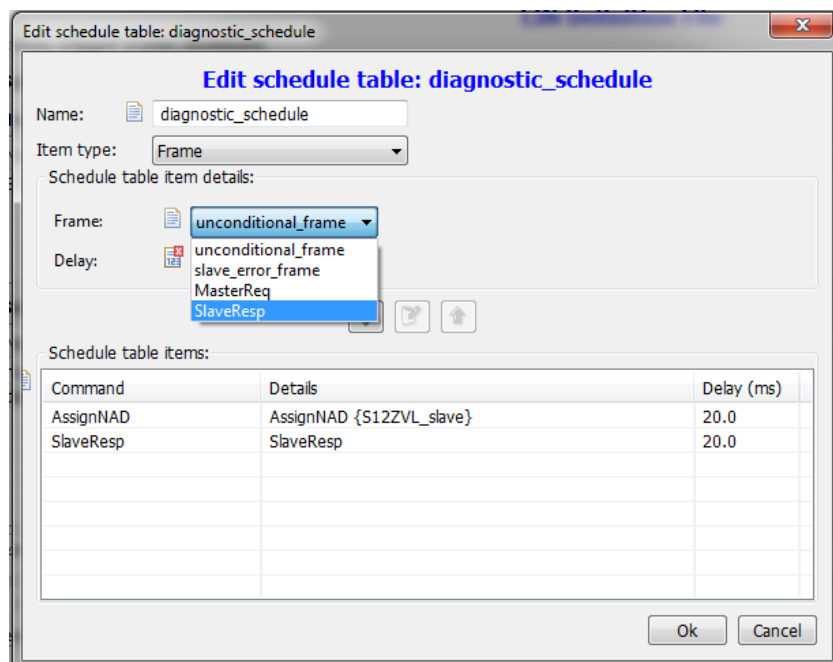


Figure 40. Selecting SlaveResp frame

Another special consideration is to select the allowable diagnostic services in the NPF file when editing the network. For each, the slave and the master.

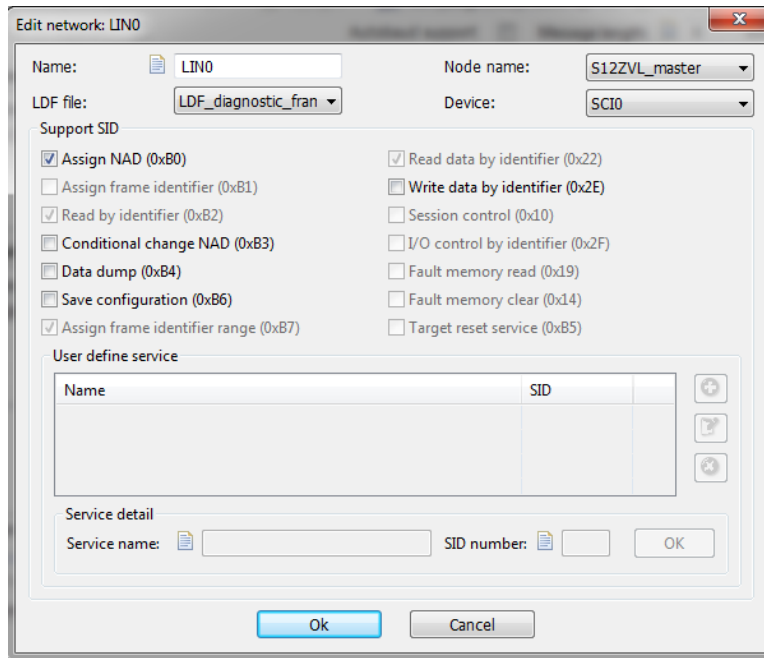


Figure 41. Network diagnostic services

For this example only Assign NAD service is selected.

The following image show the Assign NAD frame and the response of the slave. Notice that both, Assign NAD and slave response follow the diagnostic frames structures presented at the beginning of this section.

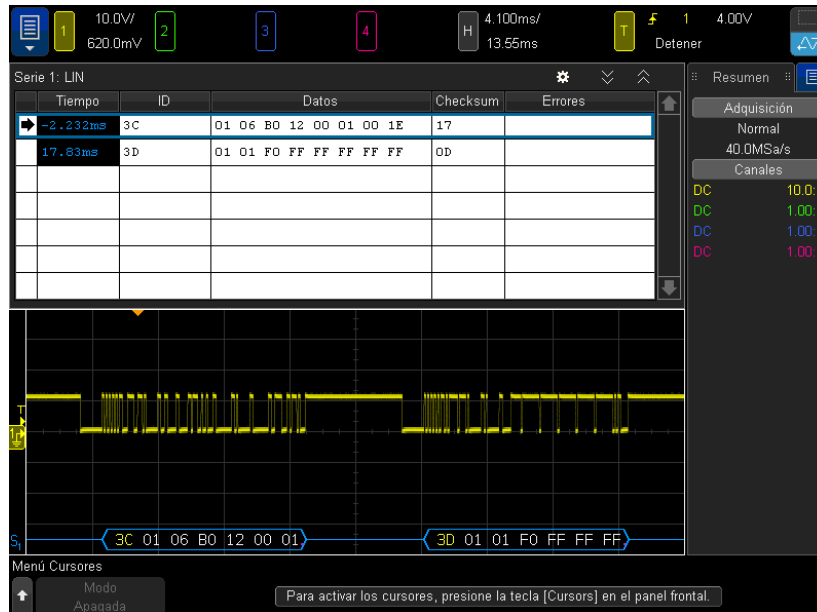


Figure 42. Diagnostic frames

The 0x3C ID frame (Master Request Frame) belongs to the AssignNAD service and the 0x3D belongs to the Slave Response. Notice that the slave response match with the positive slave response structure presented above.

8 Configuring slaves through LIN

It is often required to configure a slave once the network has been set. It is especially common on standalone LIN transceivers. According to the LIN standard the way to achieve this is using diagnostic frames that are reserved to some functionalities. This frames has a reserved SID depending on the functionality. Go to previous section in order to understand the structure of a diagnostic frame and where is the SID in the structure.

Each slave node has a unique identification number that is divided into three parameters. This number is called product identification. The three parameters that describe the product identification are the following ones:

- Supplier ID (0x0000 – 0x7FFE): The supplier ID is assigned by the LIN consortium by each of the suppliers. For example NXP supplier ID is 0x0011.
- Function ID (0x0000 – 0xFFFF): The function ID is supplier dependent.
- Variant (0x00 – 0xFF): This number depends on the product and should be changed whenever an update has been made.

Here are configuration and identification frames that can be found in the LIN standard. Review the LIN standard in order to

- Assign NAD (SID 0xB0)
Assign NAD is used to resolve conflicts with the initial NADs of the slave nodes. This frame can be used in order to set up new unique NAD to a slave.
- Assign Frame ID (SID 0xB1)
Assign Frame ID is used to set a valid protected identifier to a frame specified by its message identifier.
- Read by identifier (SID 0xB2)
Read by identifier is used to read the product identification number of a specific slave.
- Conditional Change NAD (SID 0xB3)
Conditional change NAD is used to distinguish slaves with unknown NAD from each other.
- Data Dump (SID 0xB4)
Data Dump is used for initial configuration data that a slave may require.
- Assign NAD via SNPD (SID 0xB5)
This frame is not defined by the LIN specification standard.

- Save Configuration (SID 0xB6)

Save configuration is used to tell the slave to store its current configuration in to non-volatile memory.

- Assign Frame ID Range (SID 0xB7)

Assign Frame ID range is used to configure up to four protected identifiers as frames.

Depending on what needs to be configured in the slave some of this frames may not be needed. In the slave configuration example provided in this application note it is explained how it is possible to configure a slave through LIN using the NXP's LIN stack.

The following example will illustrate how to set up the necessary diagnostic frames using the NXP's LIN stack. In this example two configuration frames are send to the slave: NAD assignation and save configuration. After each master request frame is transmitted a slave response header is transmitted in order to verify that the slave was correctly configured. In this example both nodes are S12ZVL devices. One is configured as slave and the other as master.

In order to set up a diagnostic frame in the graphic tool of the stack the following steps must be followed. First of all a diagnostic schedule must be set in the **Schedule table definition section**. This schedule table will contain all required diagnostic frames. To add diagnostic frames, click on the item type section and select one from the options. As shown in the figure below.

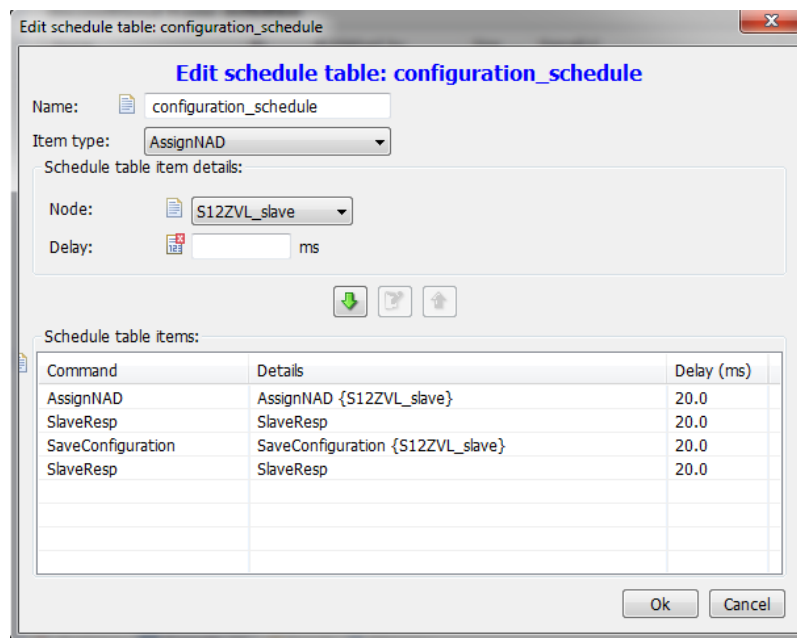


Figure 43. Adding diagnostic frames

For this example an AssignNAD and SlaveConfiguration frames were selected.

To add Slave response frames, select frame as item type and at the Frame option selection select the **SlaveResp** option. As shown in [Figure 44](#).

It is important to notice that a slave response should be issued after every diagnostic frame in order to corroborate that the slave was correctly configured.

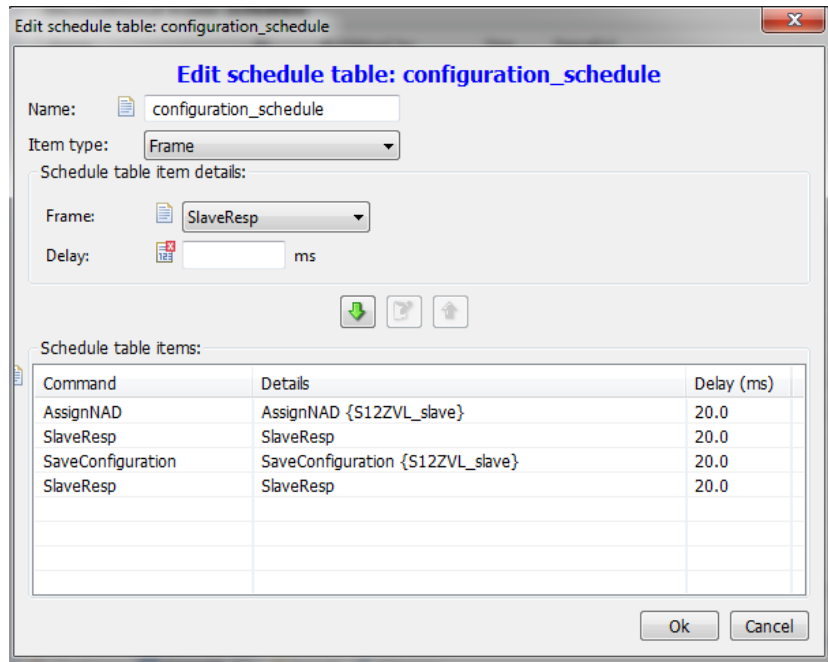


Figure 44. Assigning SlaveResponse frame

It is important that a slave response is issued after every diagnostic frame in order to corroborate that the slave has been correctly configured.

For every project generated using version 2.5.5 or previous of the LIN stack another change need to be done in the *lin_cfg.c* file of the master node. At the Schedule table initialization section all MasterReq frames that are not completely filled need to be filled, depending on the Diagnostic frame that need to be send. Here is how it is filled in this example:

```

/***** Schedule table Initialization *****/
const lin_schedule_data LIN0_configuration_schedule_data[4] = {
    {LIN0_MasterReq, 4, {0x01,0x06,0xB0,0x12,0x00,0x01,0x00,0x14}}
    , {LIN0_SlaveResp, 4, 0}
    , {LIN0_MasterReq, 4, {0x14,0x01,0xB6,0xFF,0xFF,0xFF,0xFF}}
    , {LIN0_SlaveResp, 4, 0}
};

```

For this example, the data filled match with the data necessary to send a AssignNAD and SaveConfiguration frames.

9 Go to sleep command and wake up request

9.1 Send “Go to sleep” command and “wake up” command

To send the “Go to sleep ” command the following function must be called:

- `l_ifc_goto_sleep(l_ifc_handle iii);`

Where `iii` is the name of the interface, e.g. “`l_ifc_goto_sleep(LI0);`”. This function can be called only by the Master node.

This call requests slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command. The go to sleep command will be scheduled latest when the next schedule entry is due. The `l_ifc_goto_sleep` will not affect the power mode. It is up to the application to do this. If the go to sleep command was successfully transmitted on the cluster the go to sleep bit will be set in the status register

To send the “wake up” command the following function must be called:

- `l_ifc_wake_up(l_ifc_handle iii);`

Where `iii` is the name of the interface, e.g. “`l_ifc_wake_up(LI0);`”. This function can be called by the Master node and slave nodes.

The function will transmit one wake up signal. The wake up signal will be transmitted directly when this function is called. After this function has been called, the schedule table to follow must be set with the “`l_sch_set`” function.

As explained above this driver does not actually set the slave to sleep if a go to sleep command was received. That is responsibility of the application. In this section it will be explained how the go to sleep sequence can be implemented in the MCU. Although this example is aimed to MagniV devices. The same steps can be followed by any other MCU.

Two mayor additions must be done in the LIN stack in order for it to sends the MCU to sleep and wake it up. One addition is the routine that sends the MCU to sleep and the other one is the routine that must enables that feature that wakes up the MCU when a LIN message comes.

The routine that sends the MCU to sleep must make sure to prepare all peripherals that are being used for a mode transition if is needed. For example, if the ADC is currently performing any conversion at the time the sleep commands is received. It is responsibility of the application to ensure that either the conversion is finished or to abort the conversion. Otherwise, problems may be encounter when the ADC value is needed. Refer to each of the modules in order to know if any care must be taken.

In this example a function called ***LIN_go_to_sleep_handler()*** was implemented. This function handles all the configurations that need to be done previous going to stop mode. In this case, the LIN wake up edge interruption is enabled and the SCI module is set to standby state. It is important to notice that the MCU is not being set to sleep here. The reason is that this function is called within an interruption routine of the driver. If the MCU is send to stop mode here it would never wake up until a reset is asserted.

The function of the LIN stack that calls the *LIN_go_to_sleep_handler()* can be found in the *lin_lld_sci.c* file. It is called *lin_lld_sci_set_low_power_mode()*. It is important to make sure that the *lin_lld_sci.c* file has visibility of the handler function that is being created in order for avoid errors during compilation.

NOTE

Make sure to not send the MCU to stop mode in the *lin_lld_sci_set_low_power_mode()* function or the MCU will remain hanged in stop mode and will not recover until a restart is asserted.

Until here the preparations have been done, but the MCU has not been send to stop mode. This must be done outside the *lin_lld_sci_set_low_power_mode()*. In this example it is done in the main routine. To make sure that a go to sleep command was received and processed by the MCU the *lin_goto_sleep_flg* must be checked. This flag is a status flag given by the LIN stack that allows the user to know when the MCU is ready to go to sleep. If it is set then the MCU can be send to sleep. This flag need to be cleared when waking up.

Once the flag was set the MCU must execute the proper instruction to go to stop mode. In this example, and for all MagniV devices. The routine is:

```
asm(CLI);
asm(andcc #0x7f);
asm(stop);
```

This will send the MCU to stop mode.

The routine that wakes up the MCU should be implemented in the service request interruption section of the LIN stack. As the MCU will be woke up by a LIN wake up pulse that will cause the MCU to be interrupted it is needed that this interruption is handled in order to prevent non desired functionalities of the application. The interruption routines can be found in the file *lin_lld_sci.c* in the function *lin_lld_sci_isr()*. As each core handles interruption differently it is device dependent the handling of the interruption. However, in all MCUs the steps must be similar to the ones explained in this example. In the case of MagniV devices the interruption that wake up the MCU falls in to the SCI handler as may other interruptions. Therefore, it is needed to check independently each of the interruptions that could have been set. In the SCI module of the MagniV devices the one that wakes up the MCU is the edge detection flag. The routine implemented in the example is the following one:

```
/* check if edge interrupt flag has been set and the interruption has been enabled */
if((1 == pSCI->sciasr1.abit.rxedgif)&&(pSCI->sciacr1.abit.rxedgie == 1)){
/* clear the error flag */
pSCI->sciasr1.byte |=SCIASR1_RXEDGIF_MASK;
/* wake up configuration */
pSCI->sciacr1.abit.rxedgie = 0x00; /* edge interrupt disabled for wake up*/
lin_goto_sleep_flg = 0; /* Wake up pulse received no longer need to go to sleep */
lin_goto_idle_state(); /* go to idle state as mcu just wake up */
CLEAR_PIN(P,5); /* turn on led to see when mcu is awake */
}
```


Notice that the first thing to do is to check if the desired interruption was the one that woke up the MCU. As for most of the interruption the flag must be cleared. In this case, the SCI edge interruption must be disabled to avoid being interrupted in not desired situations. In case other modules need to be re-enabled at wake up. It should be done in this routine. The next step is to clear the go to sleep flag as the MCU just came out of sleep state. Then, the LIN stack must be set to idle state using the *lin_goto_idle_state()* function in order to catch any incoming messages that follow the wake up pulse. If the LIN stack is not send to idle state then the next incoming message will not be processed due to the LIN stack will think that it is still in sleep_mode. This example includes a clearing of a signal as visual realization that the MCU is awake.

The wake up example provided in this application note includes the master and slave projects. The master project uses two buttons to send either a go to sleep command or a wake up pulse. The slave project is send to sleep if either the defined time out time passes, 4 seconds in this example, with no activity on the bus or if a go to sleep command is received, a LED is cleared indicating when the MCU has gone to stop mode. If a wake up pulse is received the MCU wakes up and the LED is turned on. The following figures illustrate the stated behavior.

The Yellow signal shows the LIN bus while the Green signal shows if the MCU is either running or in stop mode.



Figure 45. Go to sleep command received

The figure shows the reception of a go to sleep command. The green signal is at 5 V it indicating that the MCU is sleep. It was done this way because the LED lights up with a logic 0 and is turned off with a logic 1.

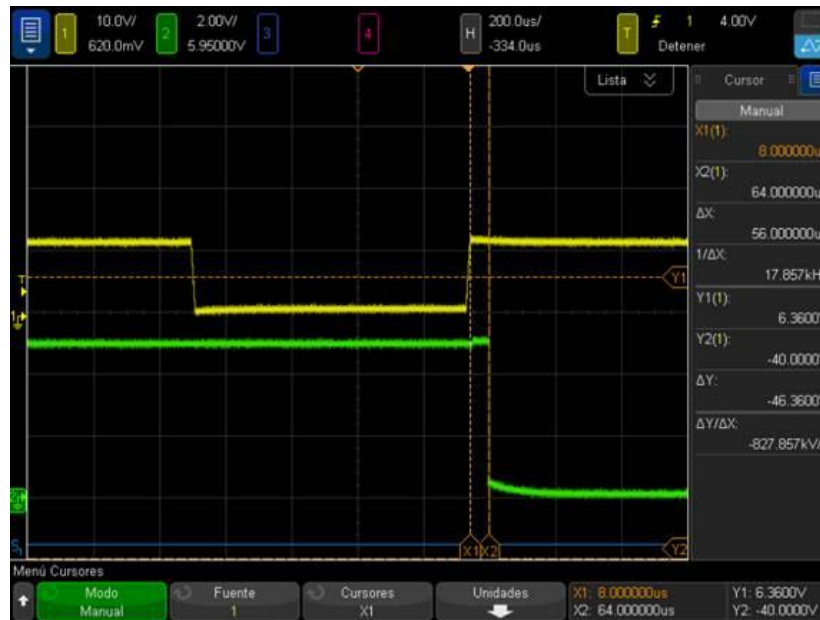


Figure 46. Wake up pulse

The figure above shows the reception of a LIN wake up pulse. It can be observed that the MCU wakes up 56 us after the pulse was received. A 0 V voltage level on the green signal indicates that the MCU is in run mode.

The [Figure 45](#) and [Figure 46](#) show the time it took to either go to sleep or to wake up.

10 References

- Package, LIN Specification. "Revision 2.2A." LIN consortium (2010).
- AN2767 Kaspar, Zdenek, Jiri Kuhn, and Roznov pod Radhostem. "LIN 2.0 Connectivity on NXP 8/16-bit MCUs Using Volcano LTP." (2004).
- Eclipse_User_Manual in the NXP LIN driver package
- LIN_User_Manual in the NXP LIN driver package
- FSL_LIN_API_Documentation in the NXP LIN driver package

How to Reach Us:

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address:
nxp.com/SalesTermsandConditions

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, and UMEMS are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, Artisan, Cortex, Jazelle, Keil, SecurCore, Thumb, TrustZone, and μ Vision are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. ARM7, ARM9, ARM11, big.LITTLE, CoreLink, CoreSight, DesignStart, Mali, mbed, NEON, POP, Sensinode, Socrates, ULINK and Versatile are trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2017 NXP B.V.

