



Please note that Cypress is an Infineon Technologies Company.

The document following this cover page is marked as "Cypress" document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content

The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers

Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

LIN Slave

5.0

PSoC 4

LIN_1

LIN Slave

Features

- Full LIN 2.2, 2.1 or 2.0 Slave Node implementation
- Compliance with ISO 17987 specification
- Compliance with SAE J2602 specification
- Compliance with LIN 1.3 specification
- Full transport layer support
- Automatic baud rate synchronization
- Automatic detection of bus inactivity
- Fully implements a Diagnostic Class I Slave Node
- Supports implementation of Diagnostic Class II and III Slave Node
- Full error detection
- Automatic configuration services
- Customizer for fast and easy configuration
- Supports import of **.ncf* and **.ldf* files, export of **.ncf* files
- Editor with syntax checking for **.ncf* and **.ldf* files

General Description

The LIN Slave Component implements a LIN 2.2, ISO 17987, and SAE J2602 compliance slave node on PSoC 4 devices. Options for LIN 1.3 and LIN 2.0 compliance are also available. This Component consists of the hardware blocks necessary to communicate on the LIN bus, and an API to allow the application code to easily interact with the LIN bus communication. The Component provides an API that conforms to the API specified by the ISO 17987 Specification.

The Component provides a good combination of flexibility and ease of use. A comprehensive customizer allows you to easily configure all parameters of the LIN Slave Node. The Component supports both NCF and LDF files to configure the LIN Slave Node.

For PSoC 4 devices only, the LIN Slave Component is certified by the C&S group GmbH based on the standard protocol and data link layer conformance tests. A complete certification report can be made available on [request](#). Contact Cypress Technical Support or check the [Component web page](#) for details.

PSoC 3 and PSoC 5LP devices are not supported with LIN v5.0. Use the LIN v4.0 Component for these devices.

Definitions

Many of the definitions given in this datasheet are from the ISO 17987:2015 and SEA J2602-1 specifications. In these cases, refer to the specified section of the ISO 17987:2015 or SEA J2602-1 specification for a proper understanding of the term.

Input/Output Connections

This section describes input and output connections for the LIN Slave.

TXD – Output

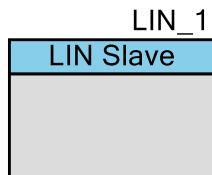
This is a digital output terminal. This terminal's signal is the data that this LIN node sends onto the LIN bus.

RXD – Input

This is a digital input terminal. This terminal's signal is the CMOS form of the signals on the physical LIN bus. Note that this terminal generally also receives any signals that come out of the TXD terminal. This is because a LIN physical layer transceiver has a built-in loop back that receives all signals on the bus, whether they are from some other LIN node, or from its own LIN node.

Schematic Macro Information

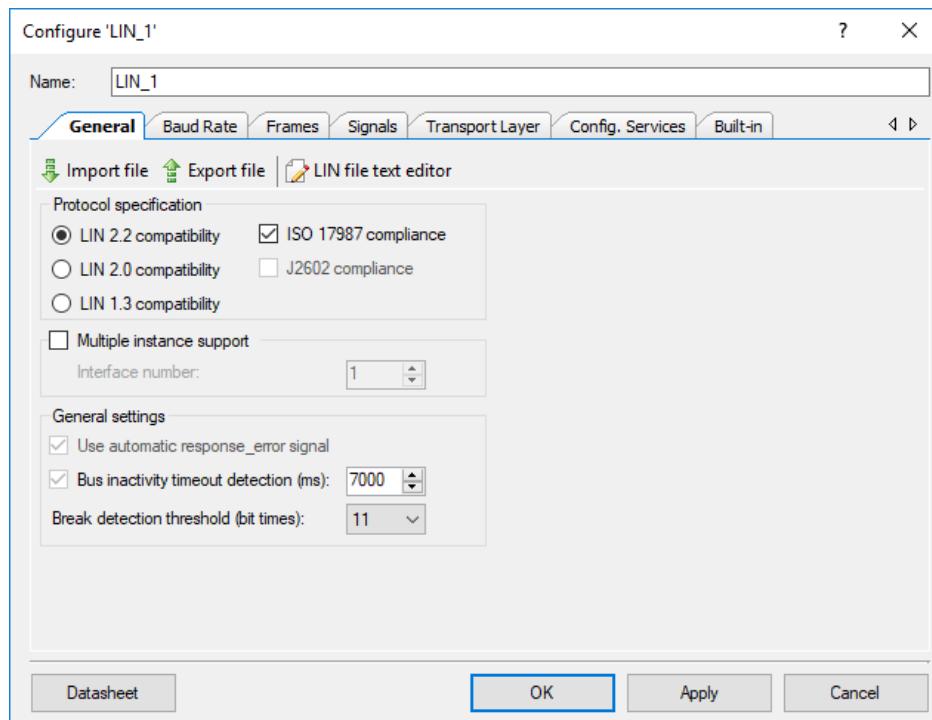
The PSoC Creator Component Catalog contains a schematic macro for the LIN Component. This macro does not have pin Components displayed. Pins must be assigned with TXD and RXD on the pins configuration page. Dedicated pins are named “\LIN_1:SCB:rx\” and “\LIN_1:SCB:tx\”. If pins are not configured, the first available SCB pins will be assigned for the LIN Component. The following shows the schematic macro with the default Component configuration.



Component Parameters

Drag a LIN Slave Component onto your design and double click it to open the **Configure LIN** dialog. The LIN Component contains parameters on several tabs.

General Tab



Protocol specification

LIN 2.2 Compatibility

This option selects whether this Component is compatible with the LIN 2.2 specification. The status of this option affects other areas of the dialog.

LIN 2.0 Compatibility

This option selects whether this Component is compatible with the LIN 2.0 specification. The status of this radio button option affects other areas of the dialog.

LIN 1.3 Compatibility

This option selects whether this Component is compatible with the LIN 1.3 specification. The status of this option affects other areas of the dialog.



ISO 17987 Compliance

This option selects whether this Component is compliant with the ISO 17987 specification. The status of this check box affects other areas of the dialog. The LIN 2.2 compatibility option must be selected.

Enable J2602 Compliance

The SAE J2602 specification is parallel to the LIN 2.x specifications. It adds restrictions to the LIN 2.0 requirements. However, there are also a few extra features that are supported by this Component that make it J2602 compliant. The status of this check box affects other areas of the dialog. The LIN 2.0 compatibility option must be selected.

Multiple instance support

This option enables/disables support for placement of two LIN Component instances in the same project. If two LIN Component instances are present on the schematic, this option must be selected for both instances, and these instances must have different interface numbers.

Interface number

This option defines the **interface number** of the current Component instance. This parameter is passed in API dynamic calls as the interface handle definition. For example, if two LIN instances: LIN_1 with **interface number** 1 and LIN_2 with **interface number** 2 are present on the schematic, the values of these definitions are:

```
LIN_1_IFC_HANDLE = 0, LIN_2_IFC_HANDLE = 1
```

General settings

Use automatic response_error signal

This check box sets the automatic error signal selection. This check box is always selected when LIN 2.0 or LIN 2.2 mode is enabled, so a 1-bit signal is automatically added in the **Signals** tab. This signal has a default name of “Response_Error.” The Component sets it automatically whenever a response error occurs. The Component also automatically clears this signal after it has been successfully sent to the master. This signal provides the response error notification to the LIN master as required by the ISO 17987 specification.

For J2602 compliance mode a 3-bits of J2602 Status ERR[2:0] are added into the first byte of each frame.

Bus Inactivity Timeout Detection

This option controls the availability of the bus inactivity feature and its value. After a specified time of bus inactivity, the corresponding status bit is set. The value of this bit can be obtained by the L_IOCTL_READ_STATUS operation of the [l_ifc_ioctl\(\)](#) function. See the [Function Description](#) section for more information.



Note The minimal timeout value in this field is restricted to 4000 milliseconds, as defined in the ISO 17987 specification. When the [LIN 1.3 Compatibility](#) option is enabled, this value could be insufficient for a LIN 1.3 inactivity timeout (25000 bit-times) at bus speed > 6000 bit/s. In such a case, the application must poll the activity bit using the [I_ifc_read_status\(\)](#) function and implement its own software inactivity timer. This approach is less optimal in terms of resource usage; however, it is directly defined in the LIN standard.

Break Detection Threshold

This option configures slave node break detection threshold. Default value is 11 dominant local slave bit times. See section 5.2.2.3. of the ISO 17987-3 specification for more information about break detection threshold selection criteria.

General Toolbar

There is a toolbar at the top of the **General** tab. This toolbar provides access to operations with files.

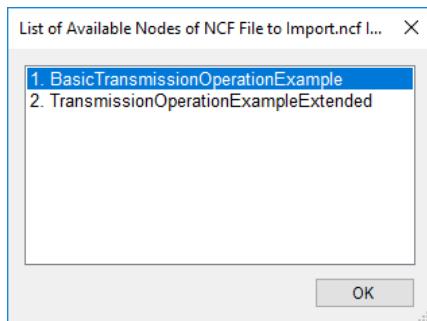


Import File

Clicking this button allows you to import a LIN Description File (LDF) or a Node Capability File (NCF). An imported file configures the customizer settings to match the configuration of the node that was selected from the list of the existing nodes of the NCF/LDF file.

If the syntax in the imported file is correct, a list of available nodes is displayed. A similar list is shown in [Figure 1](#). Choose one of the available node descriptions to import.

Figure 1. List of Available Nodes of NCF File to Import

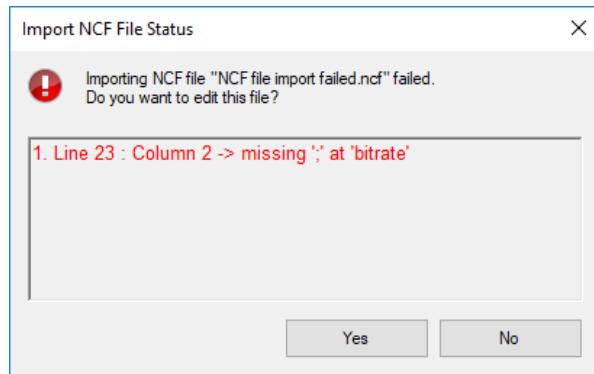


The syntax for `*.ncf` and `*.ldf` files is verified according to the LIN Node Capability Language Specification (ISO 17987-2:2015) and to the LIN Configuration Language Specification (ISO 17987-2:2015), respectively.

If the imported file contains errors, a dialog window similar to [Figure 2](#) displays. There are two options in this case: edit the imported file to correct the errors using LIN Enhanced Editor Tool (see [LIN File Text Editor](#) for more information) or cancel the import by clicking the **No** button.

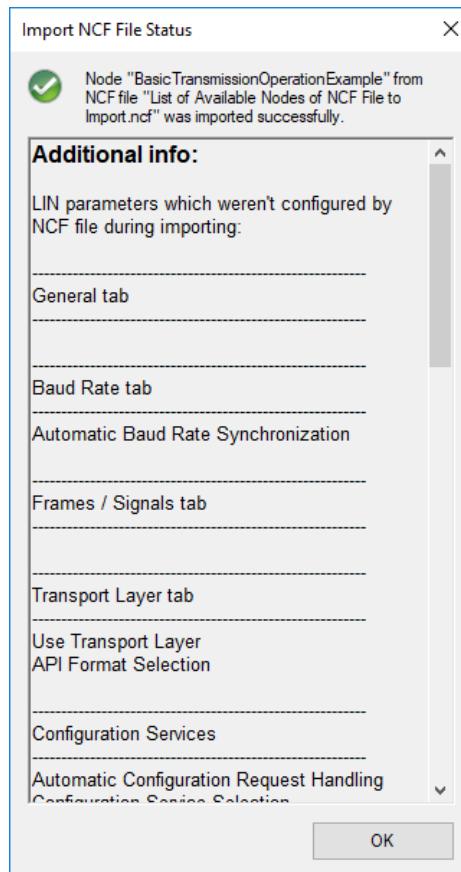


Figure 2. NCF File Import Failed



After the node to import is chosen and the import to the customizer is completed, a dialog box that describes the importing results is displayed (see [Figure 3](#)). The importing results contain the LIN Slave Component parameters that were not affected during import.

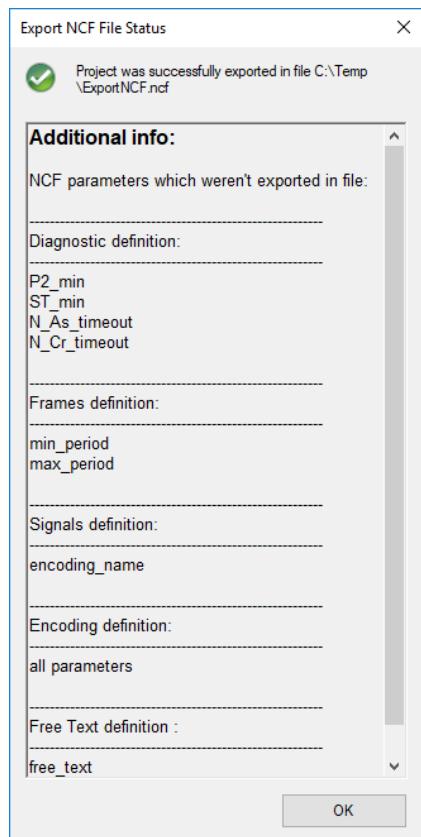
Figure 3. NCF File Import Information



Export File

This tool enables you to save information about the Component configuration into a Node Capability File (NCF).

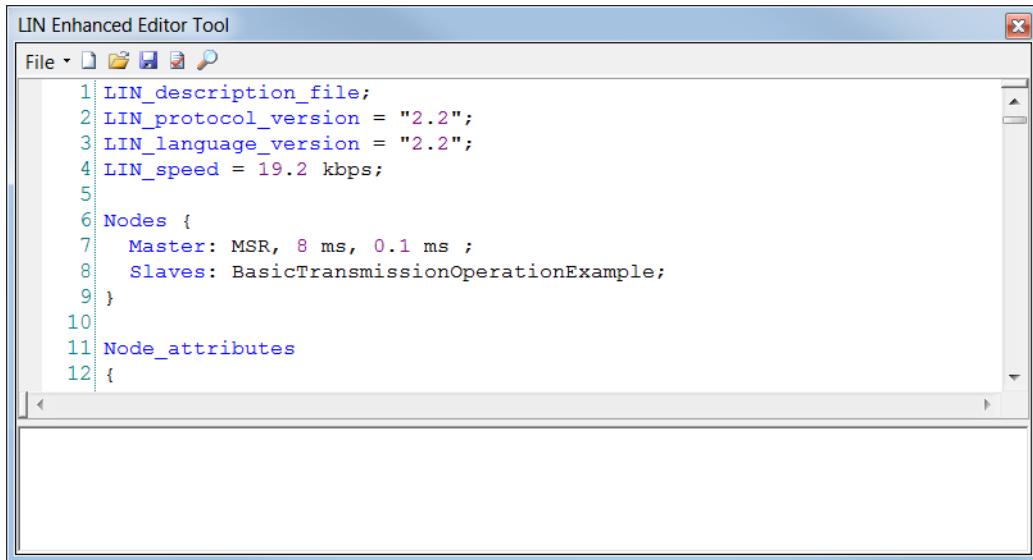
Figure 4. NCF File Export Information



LIN File Text Editor

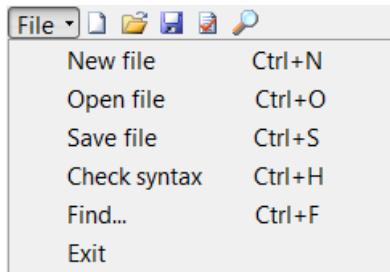
This tool is used to create, edit, and verify the syntax of the NCF/LDF file. The syntax for *.ncf files is verified according to ISO 17987-2:2015 and *LIN Node Capability Language Specification* (Revisions 2.2, 2.1, and 2.0). The syntax for *.ldf files is verified according to ISO 17987-2:2015 and *LIN Configuration Language Specification* (Revisions 2.2, 2.1, 2.0, and 1.3).

Figure 5. LIN File Text Editor Tool



There is a toolbar at the top of the **LIN Enhanced Editor Tool** window (see [Figure 6](#)).

Figure 6. LIN File Text Editor Toolbar



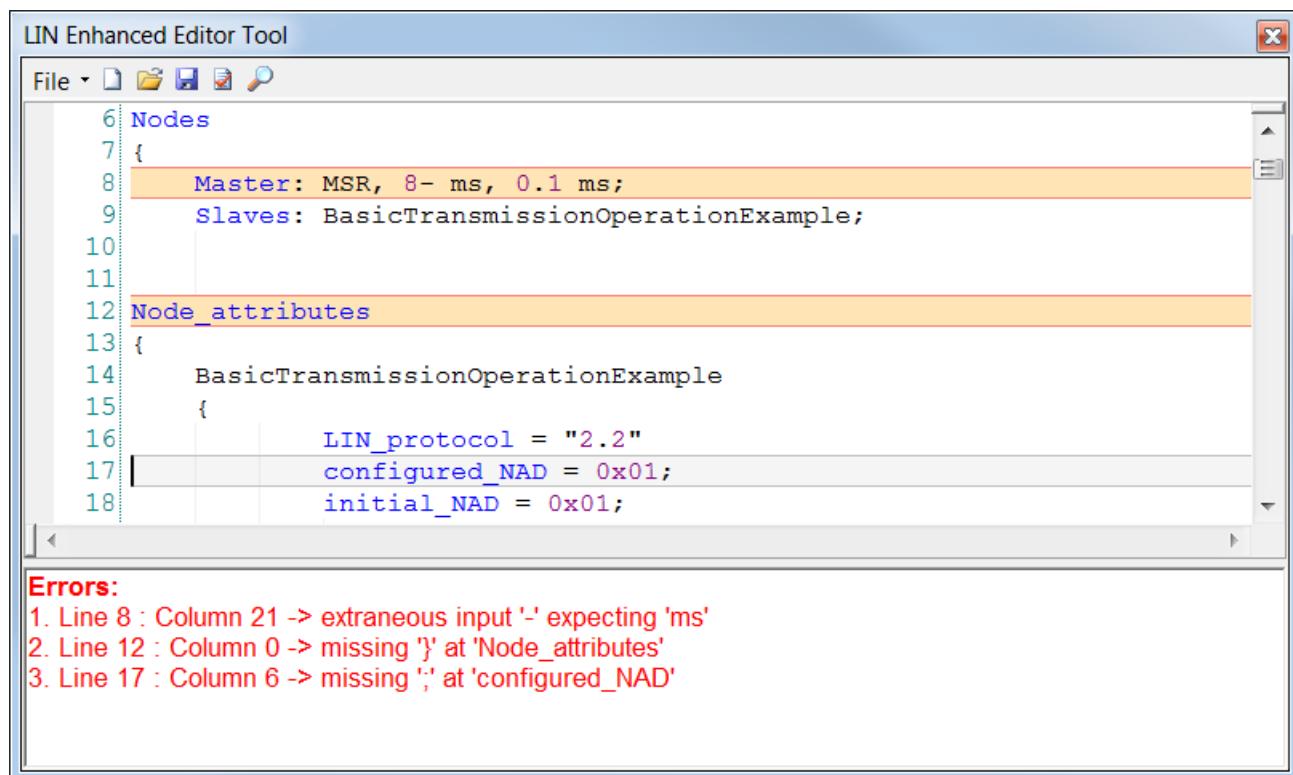
- **New File** – Creates a new file of the selected LIN file type.
- **Open File** – Opens the specified existing LIN file.
- **Save File** – Saves the created LIN file to the specified location.



- **Check Syntax** – This control allows you to check whether an ***.ncf** or ***.ldf** file syntax is correct. If there are any syntax errors, the errors are listed in the output area of the editor window with the line and column numbers of their location and a short error description (Figure 7). The code lines containing errors are highlighted in red.

Double-clicking the error line in the output area navigates to the line containing an error in file.

Figure 7. LIN File Syntax Check



The screenshot shows the LIN Enhanced Editor Tool window. The main area displays a portion of a LIN configuration file (Nodes.lcf) with syntax errors highlighted in red. Lines 8, 12, and 17 are highlighted, corresponding to the errors listed in the 'Errors' section below. The 'Errors' section contains the following list:

```

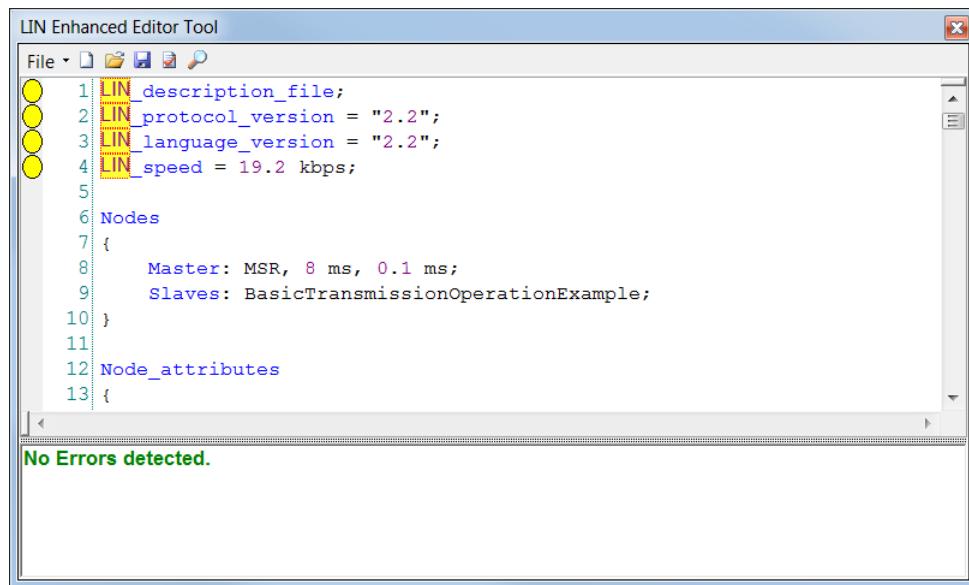
6 Nodes
7 {
8     Master: MSR, 8- ms, 0.1 ms;
9     Slaves: BasicTransmissionOperationExample;
10
11
12 Node_attributes
13 {
14     BasicTransmissionOperationExample
15     {
16         LIN_protocol = "2.2"
17         configured_NAD = 0x01;
18         initial_NAD = 0x01;

```

Errors:

1. Line 8 : Column 21 -> extraneous input '-' expecting 'ms'
2. Line 12 : Column 0 -> missing ')' at 'Node_attributes'
3. Line 17 : Column 6 -> missing ';' at 'configured_NAD'

- **Find** – This tool allows you to find the term specified in the search field in a LIN file. The **Find Next** button allocates the next match. If the **Mark Line** check box of the tool is selected, the lines containing the necessary term are labeled with yellow circles after clicking the **Find All** button. The **Style found token** check box enables or disables highlighting of the found token in yellow after clicking the **Find All** button, as shown in Figure 8. The **Clear** button removes all highlighted tokens.

Figure 8. LIN File Finding Results

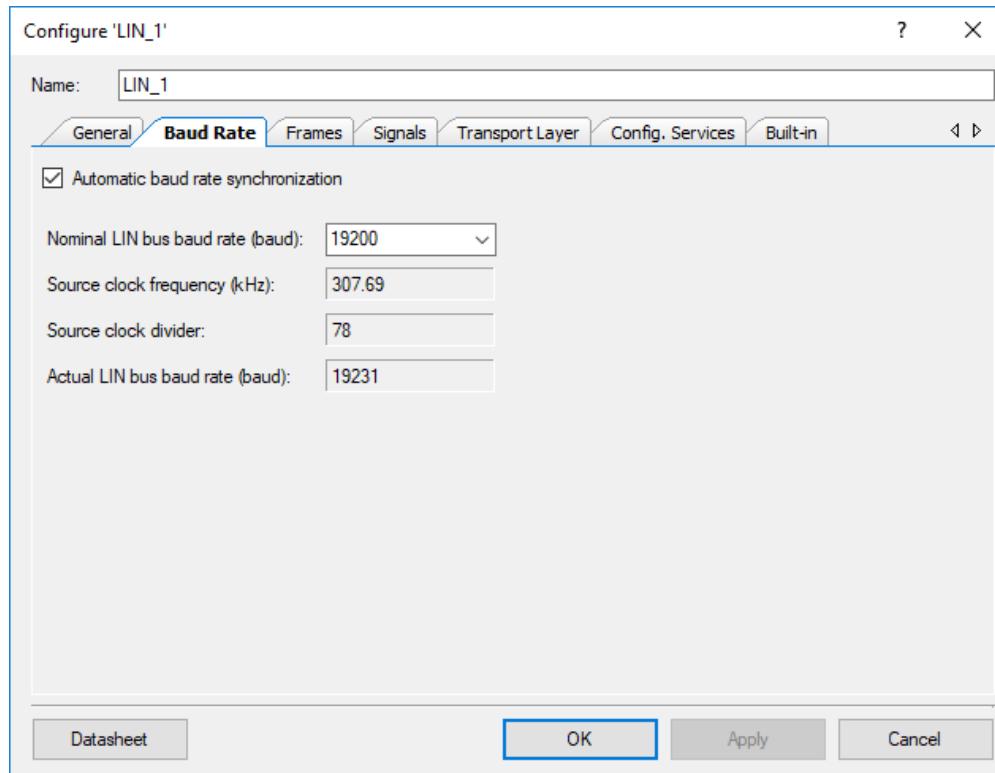
The screenshot shows the LIN Enhanced Editor Tool interface. The main window displays a configuration file with the following content:

```
1 LIN_description_file;
2 LIN_protocol_version = "2.2";
3 LIN_language_version = "2.2";
4 LIN_speed = 19.2 kbps;
5
6 Nodes
7 {
8     Master: MSR, 8 ms, 0.1 ms;
9     Slaves: BasicTransmissionOperationExample;
10 }
11
12 Node_attributes
13 {
```

Below the code editor, a message box displays the text "No Errors detected." in green.

All tools are also available in the **File** menu of the LIN Enhanced Editor Tool (see [Figure 5](#)) and through the appropriate toolbar commands.

Baud Rate Tab



Automatic Baud Rate Synchronization

This option allows you to enable or disable automatic baud rate synchronization. By default, this option is enabled.

If this option is enabled, the Component measures the exact baud rate of the bus from the sync byte field of each LIN frame header.

If this option is disabled, the Component does not measure the baud rate from the sync byte field. Instead, it receives the sync byte field as a 0x55 data byte.

As required by the ISO 17987-4 specification, LIN slave nodes with a frequency deviation of ± 1.5 percent or less do not need to use automatic baud rate synchronization to measure the sync byte field of each frame. However, if the frequency deviation of the LIN slave node is more than ± 1.5 percent, then the slave node must use automatic baud rate synchronization to measure the sync byte field of each frame.

Therefore, frequency deviation specifications must be checked for the clock source from which HFCLK clock for PSoC 4 devices is derived (this is typically the Internal Main Oscillator (IMO)).

Nominal LIN Bus Baud Rate

Enter the nominal LIN bus baud rate at which this LIN slave node must operate. The maximum value is 20000 baud and the minimum value is 1000 baud. The customizer does not allow you to select baud rates outside of this range. The values in the drop down list are 19200, 10417, 9600, and 2400. However, the combo box allows to type in any value between 1000 and 20000. If **Nominal LIN Bus Baud Rate** is modified, press the **Apply** button to get new values for the **Source Clock Frequency**, **Source Clock Divider**, and **Actual LIN Bus Baud Rate** fields.

Source Clock Frequency

This is the clock frequency, oversampled by 16, that is used for the data transmission.

Source Clock Divider

This is the value of the clock divider that is used to get the clock frequency specified in **Source Clock Frequency** from the Master clock for PSoC 3 / PSoC 5LP devices or HFCLK clock for PSoC 4 devices.

Actual LIN Bus Baud Rate

The actual value of the bus baud rate is displayed here. The LIN slave will work on this baud rate. The Master clock for PSoC 3 / PSoC 5LP devices or HFCLK clock for PSoC 4 devices value can be modified to make **Nominal LIN Bus Baud Rate** equal to **Actual LIN Bus Baud Rate**.



Frames Tab

This tab is used to configure how the LIN Slave responds to PID values that are sent by the master on the bus. The settings configured on this tab are used to correctly generate the Component API and ISR code. During operation, the LIN slave receives a PID with a frame ID in it that determines how the LIN Slave (the Component) must respond.

Index	Name	Default ID	Direction	Length	Type	Association
0	Frame1	0x01	Publish	8	Uncond...	None

Buttons on the right: Add, Delete, Up, Down, Broadcast.

Buttons at the bottom: Datasheet, OK, Apply, Cancel.

Frame Configuration Table

The configuration table contains rows and columns. Each row corresponds to one LIN frame. Note that this tab shows only “user” LIN frames. The MRF and SRF frames are supported by this Component but are not shown in this table.

There are eight possible columns in the data field, as follows:

- The fields in the **Index** column show an ordering number of each used frame. These numbers cannot be directly modified.
- The fields in the **Name** column are used to enter the name of each frame. Any string that would be valid in C code may be entered. The name of each frame must be unique.



- The fields in the **Default ID** column are used to define the frame ID that the frame will use before any configuration requests by the master. Note that these frame IDs are dynamic. In other words, the LIN master can reconfigure frame IDs at run time. You must enter a value from 0x00 to 0x3B into these cells. The values can be entered in hex or decimal format.

Note If SEA J2602 compliance is selected, the Default ID values depend on the selected NAD and will be changed when the NAD changes. The values from 0x38 to 0x3B are broadcast frames. Their Default ID will not change.

- The **Message ID** column is not normally visible. This column is only available if the **LIN 2.0 Compatibility** check box in the **General** tab of the customizer has been selected. Any 16-bit value can be entered. The value can be entered in hex or decimal format. All message ID values must be unique. Also, message ID values entered into this table should be unique for the entire LIN cluster. For example, if some other LIN slave has a frame with a message ID of 0x000F, this Component should not have any frames with a message ID of 0x000F.
- The fields in the **Direction** column define which direction the data for the frame is sent (with respect to this slave). **Publish** means a data transmission; **Subscribe** means a data reception.
- The fields in the **Length** column define how many bytes are received or sent for each frame. Values from 1 to 8, inclusive, are valid.
- The fields in the **Type** column are used to define the type of the LIN frame. There are two types of frames for LIN slave devices: **Unconditional** and **Event-Triggered**. You cannot choose the event-triggered type when the frame is a subscribe frame. In this case, this cell cannot be modified. If you change this cell from **Event-Triggered** to **Unconditional**, you must change the name of this frame to **None** in the **Association** column, if its name appears in any cells in that column.

Note If J2602 Compliance is enabled, the **Event-Triggered** frames are not accessible.

- The fields in the **Association** column are used to associate unconditional frames with event-triggered frames. An event-triggered frame must have at least one unconditional frame that is associated with it, according to the LIN specification. Therefore, the **Association** setting allows the selection of the frame name of any unconditional frames that are not already associated with an event-triggered frame. The valid values for this setting are the names of any existing unassociated unconditional frames. Only one unconditional frame can be associated with an event-triggered frame. As a result, when one of these cells has the name of an unconditional frame in it, this unconditional frame name cannot be available to any of the other rows. An event-triggered frame that is associated with an unconditional frame must have the same length and direction as the unconditional frame with which it is associated. Therefore, the name of an event-triggered frame appears only in unconditional frame rows in which these criteria apply. If you click the global **OK** button of the customizer, or if you exit this tab by clicking on another tab,



the customizer checks to make sure that there are no event-triggered frames that are not associated with any unconditional frames.

Note: The total number of frames cannot exceed 60. The total size of all frames is limited to 256 bytes.

Frames Tab Buttons

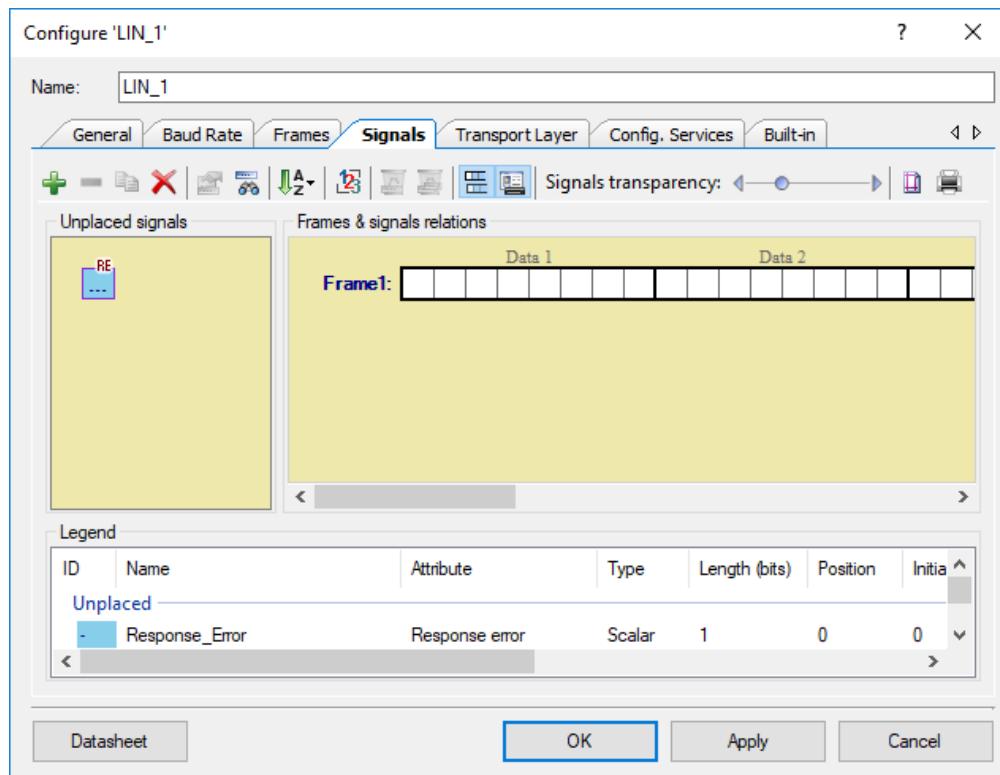
There are four buttons available on this tab.

- The **Add** button adds a new frame to the table.
- The **Delete** button deletes the currently selected frame from the table. The index number fields are changed accordingly. If a frame is deleted on this tab, any signals that are packed into it (configured with the **Signals** tab) are moved into the **Unplaced Signals** region (See [Sort Signals button](#) in the [Signals Tab](#) section).
- You can use the **Up** and **Down** buttons to reorder the **Index** number values for each frame.
- The **Broadcast** button is enabled in J2602 compliance mode. It adds four frames with Default IDs 0x38, 0x39, 0x3A and 0x3B. A frame is not added if a frame with the same ID already exists. Broadcast frames cannot be moved up in the frame table. These default IDs are not changed when the NAD is changed.



Signals Tab

This tab is used to define the “signals” that are packed into the LIN frames.



Frames & Signals relations

This graphical region of the **Signals** tab displays interactive graphics of the frames and the signals that you have defined with the customizer.

- **Frame Graphics** – One frame graphic represents each frame defined in the **Frames** tab of the customizer.
- **Signal Graphics** – Each signal graphic represents one signal defined for the LIN slave. The graphic for a signal appears as a solid bar. A signal can be placed on top of the frames using drag and drop. These signals occupy bits or bytes of the frames.

Clicking on a signal selects that signal. Rolling over a signal causes relevant information about that signal to appear in a tool-tip.

Unplaced Signals

This graphical region is a temporary region where the signals are stored after they have been added, but not placed. Signals can be moved back and forth between the **Unplaced Signals** region and the **Frames & Signals relations** region.



Note If a frame is deleted on the **Frames** tab, any signals that are packed into it (configured with the **Signals** tab) are moved into the **Unplaced Signals** region.

response_error

The 1-bit response_error signal is automatically added in the **Signals** tab of the customizer. You can change the name of the response_error signal, but you cannot delete it from the **Signals** tab.

There can be only one instance of the response_error signal and its name must be unique for this Component. The response_error signal is a Boolean signal and can be placed anywhere on a frame that is published by the LIN slave.

The purpose of this signal is to report status information to the LIN master.

For additional information about this signal see section 5.5.4 “Reporting to the Cluster” of the ISO 17987 specification.

J2602 Status Byte

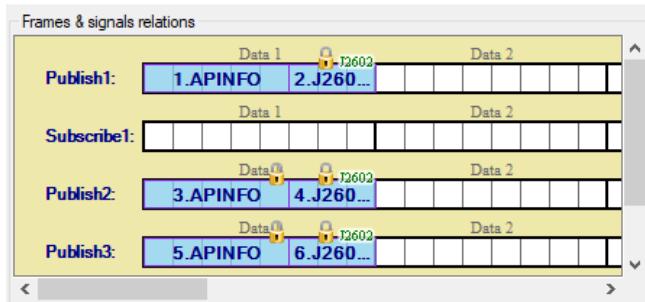
When J2602 compliance is enabled, the **J2602 Status** byte is added automatically to the frames with **Publish** direction. It consists of five bits of **APINFO** and three bits of **ERR** fields.

Three bits of **ERR** field are always present with default name “**J2602_Status**.” They are handled by Component and there is no read nor write API. The application can only call [LIN_j2602Status_Reset\(\)](#) to clear all error states and indicate that the LIN Slave needs to be configured by the LIN Master. The lock sign indicates that signals cannot be changed.

ERR2	ERR1	ERR0	Fault State	Priority
0	0	0	No Detected Fault	0 (lowest)
0	0	1	Reset	1
0	1	0	Reserved	2
0	1	1	Reserved	3
1	0	0	Data Error	4
1	0	1	Checksum Error	5
1	1	0	Byte Field Framing Error	6
1	1	1	ID Parity Error	7 (highest)



Five bits of **APINFO** field are defined by application. APINFO fields of all published frames must be identical. It is allowed to add or edit signals in APINFO field of first published frame. These signals will be cloned to rest of publish frames. Lock sign indicates that signals cannot be changed.



Note The initial value of J2602_Status byte, after component initialization, is zero, regardless on loaded configuration.

Signals Toolbar

There is a toolbar at the top of the **Signals** tab. This toolbar provides an easy way to manage the signals on the tab.



1. Add/Delete buttons

The **Add Signal** button adds a signal to the **Unplaced Signals** region. The **Delete Signal** button removes selected signals from the Component. The **Delete All Signals** button removes all existing signals.

2. Clone Signal button

3. Clones selected signal to the Unplaced Signals region. **Signal Properties** button

This control opens the **Signal Properties** window for the selected signal. This window can be used to change the properties for the signal. Note that the properties window for a signal can also be accessed by double clicking on a signal.

4. Find Signal button

This button allows you to search for a certain signal.

5. Sort Signals button

This button sorts the signals in the **Unplaced Signals** region. Signals can be sorted by Name, Length, or Type.

6. Renumber Signals button

This button renames the signal index values in ascending order.

7. Move buttons



The **Unplace Signal** button moves the selected signal from the **Frames & Signals relations** region to the **Unplaced Signals** region.

The **Unplace All Signals** button moves all signals to the **Unplaced Signals** region.

8. Show/Hide Event-triggered frames button

This button allows you to show or hide the frames graphics that correspond to event-triggered frames in the **Frames & Signals relations** region.

9. Show/Hide Legend button

This button allows you to show or hide the legend area describing the signals' properties.

10. Signals Transparency slider

This slider sets the transparency for signals graphics.

11. Page setup and Print buttons

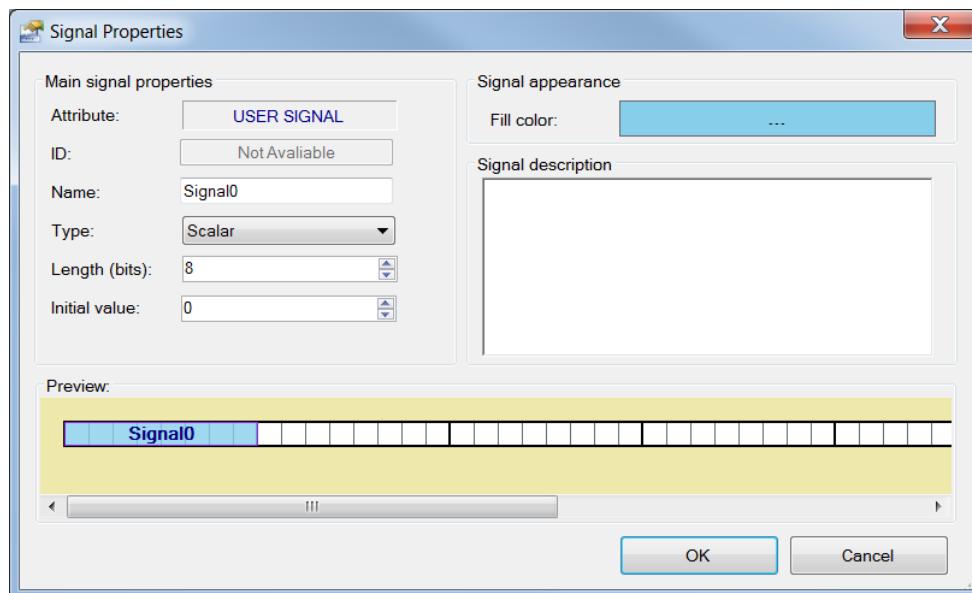
These buttons allow to print out the **Frames & Signals relations** region.

Signal Properties Window

Adding Signals

There is an **Add Signal** button on the tool bar. This button causes a new window to appear with signal property options that can be configured (see [Figure 9](#)). After the properties have been configured, a new signal is added. The various signal properties that can be configured on this window are described in this section.

Figure 9. Signal Properties Window



Name

The **Name** property is used to choose the name of the signal. The default signal name is Signal x , where ‘ x ’ is equal to the index number of the signal. The name entered for the signal must be a valid symbol name in C code.

Note If several signals have the same name:

- These signals must have equal “Type”, “Length”, and “Initial Value” properties.
- One Frame cannot accept two or more duplicates.
- Signals can have duplicates only on Data Frames which “Direction” property is set to “Publish.”

Type

This property is used to select the type of the signal. There are two types of signal, as defined in the ISO 17987 specification. A **Scalar** signal is 1 to 16 bits in length and a **ByteArray** signal is 1 to 8 bytes in length.

Length

This property is used to select the length of the signal. Scalar signals can have a length of 1 to 16 bits. A ByteArray signal can have a length of 1 to 8 bytes.

Initial Value

This property is used to select the initial value for the signal. This value must be entered in decimal format.

Fill Color

This control is used to select a color for the signal graphic.

Signal Description

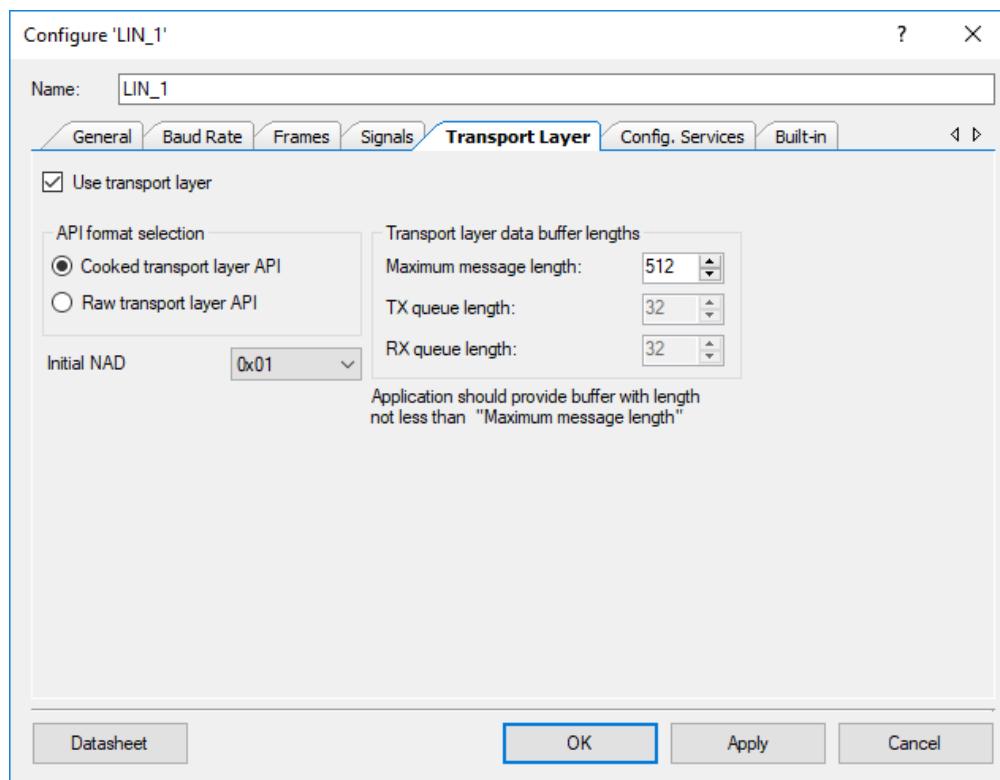
This property can be used to enter any relevant description or other information related to the signal.

Preview

This graphical area shows what the signal will look like when it is added.



Transport Layer Tab



Use Transport Layer

If the **Use Transport Layer** check box is not selected, the slave node will not support the Transport Layer. If it is selected, the slave node Component will support the Transport Layer. See the ISO 17987-2 specification for detailed information on the Transport Layer.

API Format Selection

This control is used to select the format for the Transport Layer API functions. There is a **Cooked Transport Layer API** option and a **Raw Transport Layer API** option. **Typically, the cooked format is recommended for LIN slave applications.** Raw API is intended for LIN gateway applications.

The cooked format is used to send and receive Transport Layer messages using just one API function for each message. The raw format is used to send or receive each frame that makes up a Transport Layer message using one API function call for each frame.

The two formats of the Transport Layer API are defined by ISO 17987-5 in Transport layer section.



Initial NAD

This field is used to select the Network Address (NAD) of the slave node. The NAD is used in MRF and SRF frames to address one particular slave node in a cluster. Note that this field is used to select the Initial NAD for the node. The NAD of a slave node can change at run time.

By default, the Initial NAD value can be in the range from 0x01 to 0xFF. The NAD value of 0x00 is reserved for a “Go To Sleep” command. The NAD value of 0x7E is reserved as a “Functional NAD” which is used for diagnostic services. The NAD value of 0x7F is reserved as a “wildcard” NAD. Therefore, the customizer restricts you from entering 0x00, 0x7E, or 0x7F into this field.

If the J2602 compliance check box is checked, the Initial NAD value on the Transport Layer Tab is restricted to 0x60 to 0x6F. The default value is 0x60. The initial value range is further restricted based on the number of frames that are used on the **Frames** tab of the customizer. See [Table 1](#) for more information.

Table 1. Initial NAD Restriction Based on the Number of Frames Used in Slave Node

Number of Frames	Available Initial NAD Values
1 to 4	0x60 to 0x6F
5 to 8	0x60, 0x62, 0x64, 0x66, 0x68, 0x6A, 0x6C, 0x6E, 0x6F
9 to 16	0x60, 0x64, 0x68, 0x6E, 0x6F
More than 16	0x60, 0x6E, 0x6F

Maximum Message Length

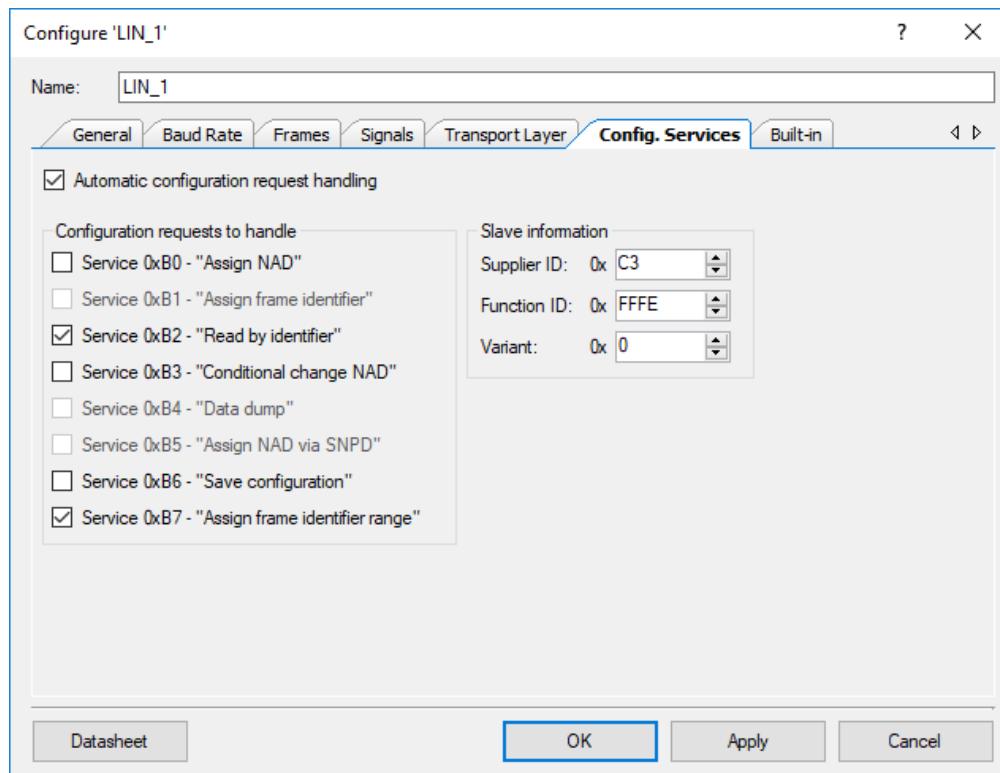
This property is used to select the maximum Transport Layer message length that this slave node supports. The minimum value is 6, because there are up to six Transport Layer message data bytes in messages that use only one frame. This Component only supports Transport Layer messages with lengths up to 4095 bytes. Note that the actual Transport Layer message buffer is located in the application code of the node.

TX Queue Length/RX Queue Length

These properties are only applicable when the **Raw Transport Layer API** format is selected. When using the raw API format, there is a message “queue” that buffers the frame response data that is being sent or received. If the slave cannot update the queues very quickly, then the queue lengths should be made longer. If the slave can update the queues very quickly, then the queues can be made shorter to decrease RAM memory use. The Component supports queue lengths from 8 to 2048 with 8-byte steps. The default size of each queue is 32 bytes.



Configuration Services Tab



The ISO 17987 specification defines Configuration Service requests that the slave must support (some are mandatory and some are optional with regard to the ISO 17987 specification). This Component supports all mandatory requests and some optional service requests.

There are eight total configuration service requests (0xB0 to 0xB7). There is a list of these services in Table 6 of the ISO 17987-3 specification. You have the option of disabling or enabling each of the supported services individually. The configuration service requests are described in section 6.3.6 Node configuration services of the ISO 17987-3 specification.

Automatic Configuration Request Handling

The Component is designed so that it automatically handles configuration service requests. In other words, you do not have to use any API or application code to service these requests from the master. However, you can disable this automatic handling and handle these requests with your own custom application code.

To simplify this option, there is an **Automatic Configuration Request Handling** check box on this tab. If the box is checked, all of the other options on the tab are available. If the check box is not checked, then all of the other options on the tab are disabled.

Any service that is enabled in this tab is automatically handled by this Component. Whenever any of these automatically handled requests occur during LIN bus operation, the corresponding MRF and SRF frames will not be available to the application through the Transport Layer API. If a service request is not automatically handled (that is, if it is not enabled on this tab), then the



corresponding MRF and SRF frames of the configuration service request must be received or sent by the application using the Transport Layer API.

Configuration Service Selection

Each of the supported configuration service requests is listed on the tab with a check box. You can individually select the services that you want to be automatically handled.

- Service 0xB0 – “Assign NAD”

This is an optional service in the ISO 17987 specification.

This is a service request where a new NAD value is assigned to the slave node.

This service request is not likely to be needed for this Component, due to the highly-programmable nature of PSoC devices. The PSoC can easily configure its NAD to a desired value after it boots up, and probably does not need the LIN master to request a NAD change.

- Service 0xB1 – “Assign Frame Identifier”

This is an obsolete service in the ISO 17987 specification. It is only available if the **LIN 2.0 Compatibility** checkbox has been selected on the **General** tab of the customizer.

This configuration service request is used to change the frame ID value for a frame to which this slave node responds.

This service is described in the LIN 2.0 specification in section 2.5.1. This service is available in this Component for backwards compatibility purposes.

- Service 0xB2 – “Read by identifier”

This configuration service request is mandatory according to the ISO 17987 specification.

This request is used to allow the LIN master to read the slave's identification information (Supplier ID, Function ID, Variant). This Component supports the LIN Product Identification version, Serial number, optional NCF/LDF version for ISO 17987 nodes, and 'Message ID' parameter for LIN 2.0 nodes of this request.

- Service 0xB3 – “Conditional Change NAD”

This is an optional service in the LIN 2.2 and ISO 17987 specifications.

This is very similar to the Assign NAD configuration service. One major difference is that this service uses the slave's current (volatile) NAD instead of the initial (nonvolatile) NAD. When this request occurs, the slave does some logic processing on the data bytes received from the master and only updates its current (volatile) NAD if the result of the processing is zero.



- Service 0xB4 – “Data Dump”

This service is reserved for initial configuration of a slave node by the slave node supplier and the format of this message is application specific and is not supported by this Component. Received data is transferred to application with transport layer.

- Service 0xB5 – “Assign NAD via SNPD” (Targeted Reset)

“Assign NAD via SNPD” (0xB5) service is not supported by the ISO 17987 specification. However, when the **Enable J2602-1 Compliance** check box is selected on the **General** tab, this service (0xB5) has a different meaning: Targeted Reset, which is supported by the Component.

If a Targeted Reset request is processed by this slave, a flag is set in the `L_IOCTL_READ_STATUS` operation of the `l_ifc_ioctl()` function to let the application know that a Targeted Reset should occur. Refer to the [API description](#) for more information.

- Service 0xB6 – “Save Configuration”

This is an optional service request in the ISO 17987 specification.

The slave device can save its configuration data (NAD value and PID values) in nonvolatile memory (flash). However, the application code must implement the actual flash writing operations.

When this configuration service request occurs, the Save Configuration flag in the status returned by the `l_ifc_read_status()` API function is set. This lets the application know that it must save its current LIN slave node configuration information to nonvolatile memory (flash).

- Service 0xB7 – “Assign frame identifier range”

This is a mandatory configuration service request in the ISO 17987 specification.

This service allows the LIN master to change the volatile frame PID values for the slave’s frames.

Slave Information

If you have checked the **Automatic Configuration Request Handling** check box, three fields become available.

The fields are **Supplier ID**, **Function ID**, and **Variant**. The Supplier ID is a 16-bit value, but its valid range is from 0x0000 to 0x7FFE. The Function ID is also 16 bits, and its valid range is 0x0000 to 0xFFFF. The Variant is 8 bits and its valid range is from 0x00 to 0xFF.

These values are used in the configuration service requests to differentiate between the different slave nodes in a LIN cluster. So, these values act as a type of slave address in some ways.



Clock Selection

PSoC Creator calculates the needed frequency and clock source and generates the resource needed for implementation. The clock tolerance must be within ± 1.5 percent when the **Automatic Baud Rate Synchronization** option is disabled and ± 14 percent when enabled. A warning will be displayed if the clock cannot be generated within this limit. In this case, you should modify the HFCLK clock source for PSoC 4 devices in the DWR.

Placement

Placement of two Components is allowed for a PSoC 4 design with the assistance of the LIN_Dynamic Component.

LIN_Dynamic Component

The LIN Component is linked with a hidden design-wide LIN_Dynamic Component, which is always present in a design to support the placement of multiple instances of the LIN Component.

The LIN_Dynamic Component routes all LIN dynamic API calls to the appropriate instance of the Component. Dynamic API functions consist of a single switch statement that calls the corresponding static API function depending on the passed parameters:

- `I_signal_handle` for signal interaction functions
- `I_flag_handle` for notification functions
- `I_ifc_handle` for interface management and transport layer functions

Notes:

- The `I_signal_` and `I_flag_` handles are defined in the `LIN.h` file and enumerated in range (0..127) for the first LIN instance and in range (128..255) for the second instance.
- The `I_ifc_handles` are also defined in the `LIN.h` file. They are equal to 0 for the first LIN instance and 1 for second instance.

Additionally, the LIN_Dynamic Component contains an implementation of `I_sys_init()` API function, which is common for all LIN Component instances as defined by the LIN Specification.



LIN Version and Updates

The LIN_Dynamic version must be the same as LIN Component used in the design. Therefore, both the LIN and the LIN_Dynamic Components must be updated synchronously.

The LIN_Dynamic Component is also shown in the Component Update Tool because of its nature as a design-wide Component. If you do not have a LIN Component in your design, then no action is required, and the LIN_Dynamic Component is inactive and colored gray.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the Component using software. The following tables list and describe the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name “LIN_1” to the first instance of a Component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “LIN.”

Core API Functions

Initialization Subgroup

Function	Description
<code>I_bool I_sys_init()</code>	Initializes the LIN core.

`I_bool I_sys_init()`

Description: Initializes the LIN core. If the Automatic Baud Rate Synchronization parameter is enabled in the Configure dialog, then this function saves the initial SYSCLK-to-UARTCLK divider’s value, calculated by PSoC Creator. If the parameter is not enabled, this function does nothing.

Static Prototype: `I_bool I_sys_init(void)`

Return Value: Always returns zero.



Signal Interaction Functions Subgroup

In all static signal API calls that follow, the “sss” is the name of the signal, for example, `I_u8_rd_EngineSpeed()`. For dynamic signal API calls that follow, the “sss” is a signal handle, as defined in [Application Programming Interface](#).

Function	Description
<code>I_bool_rd()</code>	Reads and returns the current value of the signal for one-bit signals.
<code>I_u8_rd()</code>	Reads and returns the current value of the signal for signals of two to eight bits.
<code>I_u16_rd()</code>	Reads and returns the current value of the signal for signals of 9 to 16 bits.
<code>I_bytes_rd()</code>	Reads and returns the current values of the selected bytes in byte array signal.
<code>I_bool_wr()</code>	Sets the current value of the signal for one-bit signals.
<code>I_u8_wr()</code>	Sets value for signals of two to eight bits.
<code>I_u16_wr()</code>	Sets value for signals of 9 to 16 bits.
<code>I_bytes_wr()</code>	Sets values of the selected bytes in byte array signal.

`I_bool_rd()`

Description: Reads and returns the current value of the signal for one-bit signals. If an invalid signal handle is passed into the function, no action is taken, function returns 0x00.

Static Prototype: `I_bool I_bool_rd_sss(void)`

Dynamic Prototype: `I_bool I_bool_rd(I_signal_handle sss)`

Parameters: `sss`: Signal handle of the signal to read.

Return Value: Returns the current value of the signal.

`I_u8_rd()`

Description: Reads and returns the current value of the signal. If an invalid signal handle is passed into the function, no action is taken, function returns 0x00.

Static Prototype: `I_u8 I_u8_rd_sss(void)`

Dynamic Prototype: `I_u8 I_u8_rd(I_signal_handle sss)`

Parameters: `sss`: Signal handle of the signal to read

Return Value: Returns the current value of the signal.

I_u16_rd()

Description:	Reads and returns the current value of the signal. If an invalid signal handle is passed into the function, no action is taken, function returns 0x00.
Static Prototype:	<code>I_u16 I_u16_rd_sss(void)</code>
Dynamic Prototype:	<code>I_u16 I_u16_rd(I_signal_handle sss)</code>
Parameters:	Sss: Signal handle of the signal to read
Return Value:	Returns the current value of the signal.
Side Effects:	This function does not guarantee that the data bytes that are read are atomic. If it is necessary for the data bytes to be atomic, then the application must ensure that this is the case.

I_bytes_rd()

Description:	Reads and returns the current values of the selected bytes in the signal. The sum of the start and count parameters must never be greater than the length of the byte array. Note that when the sum of start and count is greater than the length of the signal byte array, an accidental data is read.
	If an invalid signal handle is passed into the function, no action is taken.
	Assume that a byte array is 8 bytes long, numbered 0 to 7. Reading bytes from 2 to 6 from a user-selected array requires start to be 2 (skipping byte 0 and 1) and count to be 5. In this case, byte 2 is written to user_selected_array[0] and all consecutive bytes are written into user_selected_array in ascending order.
Static Prototype:	<code>void I_bytes_rd_sss(I_u8 start, I_u8 count, I_u8* const data)</code>
Dynamic Prototype:	<code>void I_bytes_rd(I_signal_handle sss, I_u8 start, I_u8 count, I_u8* const data)</code>
Parameters:	<p>sss: Signal handle of the signal to read</p> <p>start: First byte to read from</p> <p>count: Number of bytes to read</p> <p>data: Pointer to array, in which the data read from the signal is stored</p>
Side Effects:	This function does not guarantee that the data bytes that are read are atomic. If it is necessary for the data bytes to be atomic, then the application must ensure that this is the case.

I_bool_wr()

Description:	Writes to the one-bit signal. If an invalid signal handle is passed into the function, no action is taken.
Static Prototype:	<code>void I_bool_wr_sss(I_bool v)</code>
Dynamic Prototype:	<code>void I_bool_wr(I_signal_handle sss, I_bool v)</code>
Parameters:	<p>sss: Signal handle of the signal to write</p> <p>v: Value of the signal to be set</p>



I_u8_wr()

Description: Writes to the signal of two to eight bits. If an invalid signal handle is passed into the function, no action is taken. If the value written to the signal is larger than the signal's length, the higher bits are discarded.

Static Prototype: void I_u8_wr_sss(I_u8 v)

Dynamic Prototype: void I_u8_wr(I_signal_handle sss, I_u8 v)

Parameters:
sss: Signal handle of the signal to write
v: Value of the signal to be set

I_u16_wr()

Description: Writes to the signal of 9 to 16 bits. If an invalid signal handle is passed into the function, no action is taken. If the value written to the signal is larger than the signal's length, the higher bits are discarded.

Static Prototype: void I_u16_wr_sss(I_u16 v)

Dynamic Prototype: void I_u16_wr(I_signal_handle sss, I_u16 v)

Parameters:
sss: Signal handle of the signal to write;
v: Value of the signal to be set.

Side Effects: This function does not guarantee that the data bytes that are written will be read atomically by the LIN master. If it is necessary for the data bytes to be atomic, then the application must ensure that this is the case.

I_bytes_wr()

Description: Writes the current value of the selected bytes to the signal specified by the name sss. The sum of start and count must never be greater than the length of the byte array, although the device driver may choose not to enforce this in run time. Note that when the sum of start and count is greater than the length of the signal byte array an accidental memory area is affected.

If an invalid signal handle is passed into the function, no action is taken.

Assume that a byte array signal is 8 bytes long, numbered 0 to 7. Writing byte 3 and 4 of this array requires start to be 3 (skipping bytes 0, 1, and 2) and count to be 2. In this case, byte 3 of the byte array signal is written from user_selected_array[0] and byte 4 is written from user_selected_array[1].

Static Prototype: void I_bytes_wr_sss(I_u8 start, I_u8 count, const I_u8* const data)

Dynamic Prototype: void I_bytes_wr(I_signal_handle sss, I_u8 start, I_u8 count, const I_u8* const data)

Parameters: sss: Signal handle of the signal to write

start: First byte to write to

count: Number of bytes to write

data: Pointer to array, in which the data to transmit to LIN master is located

Side Effects: This function does not guarantee that the data bytes that are written are read atomically by the LIN master. If it is necessary for the data bytes to be atomic, then the application must ensure that this is the case.

Notification Functions Subgroup

Notification flags are used to synchronize the application program with the LIN core. The flags are automatically set by the LIN core and can only be tested or cleared by the application program. A notification flag can correspond with a signal, a signal in a particular frame (in the case that the same signal is packed into multiple frames), or a frame. A flag is set by this Component when the corresponding signal or frame is successfully sent or received.

In all of the following flag API routines the “fff” is the name of the flag, for example, I_flg_tst_RxEngineSpeed(). For the dynamic flag API routines the “fff” is a signal handle, as defined earlier in [Application Programming Interface](#).

Function	Description
I_flg_tst()	Returns a boolean indicating the current state of the flag.
I_flg_clr()	Sets the current value of the flag to zero.



I_flg_tst()

Description:	This function returns current state of the flag specified by the name “fff.” It returns false if the flag is cleared and true otherwise. If this routine returns a “true” value, then it indicates that the corresponding signal or frame has been successfully sent or received.
Static Prototype:	<code>I_bool I_flg_tst_fff(void)</code>
Dynamic Prototype:	<code>I_bool I_flg_tst(I_flag_handle fff)</code>
Parameters:	fff: Name of the flag handle
Return Value:	Returns a C boolean indicating the current state of the flag specified by the name “fff”. false: The flag is cleared; true: The flag is not cleared.

I_flg_clr()

Description:	Clears the flag that is specified by the name “fff”. This routine should be used to clear a flag after it has been tested (after <code>I_flg_tst()</code> API). The Component does not automatically clear notification flags. This routine is the only way that a notification flag can be cleared.
Static Prototype:	<code>void I_flg_clr_fff(void)</code>
Dynamic Prototype:	<code>void I_flg_clr(I_flag_handle fff)</code>
Parameters:	fff: Name of the flag handle

Interface Management Functions* Subgroup

These calls manage the specific interfaces (the logical channels to the bus). Each interface is identified by its interface name, denoted by the “iii” extension for each static API call, for example, `I_ifc_init_MyLinIfc()`. For static prototypes, the interface name is the same as the Component instance name. This Component supports a maximum of two interfaces. Therefore, there will never be more than two valid identifiers for “iii.” For dynamic prototypes, iii is a number defined as `MyLinIfc_IFC_HANDLE`. It is equal to 0 for a single instance and equal to 0 or 1 when both interfaces are present in the schematic.

Function	Description
<code>I_ifc_init()</code>	Initializes the LIN Slave Component.
<code>I_ifc_wake_up()</code>	Transmits one wakeup signal.
<code>I_ifc_ioctl()</code>	Controls functionality beyond the specification.
<code>I_ifc_rx()</code>	The LIN Slave Component calls this API routine automatically.
<code>I_ifc_tx()</code>	The LIN Slave Component calls this API routine automatically.
<code>I_ifc_aux()</code>	The LIN Slave Component calls this API routine automatically.

Function	Description
<code>l_ifc_read_status()</code>	Returns the status of the specified LIN interface.

l_ifc_init()

Description: `l_ifc_init()` initializes the LIN Slave Component instance that is specified by the name “iii.” It sets up internal functions such as the baud rate and starts up digital blocks that are used by the LIN Slave Component. This is the first call that must be performed, before using any other interface-related LIN Slave API functions.

Static Prototype: `l_bool l_ifc_init_iii(void)`

Dynamic Prototype: `l_bool l_ifc_init(l_ifc_handle iii)`

Parameters: `iii`: Name of the interface handle

Return Value: The function returns zero if the initialization was successful and nonzero if it failed.

l_ifc_wake_up()

Description: This function transmits one wakeup signal. The wakeup signal is transmitted directly when this function is called. When you call this API function, the application is blocked until a wakeup signal is transmitted on the LIN bus. The `CyDelayUs()` function is used as the timing source. The delay is calculated based on the clock configuration entered in PSoC Creator.

Static Prototype: `void l_ifc_wake_up_iii(void)`

Dynamic Prototype: `void l_ifc_wake_up(l_ifc_handle iii)`

Parameters: `iii`: Name of the interface handle



[_ifc_ioctl\(\)](#)

Description: This API controls functionality that is not covered by the other API calls. This function is used to control this Component in device-specific ways.

For the operations that are supported by this function, refer to the [Component Parameters](#) section.

Static Prototype: `l_u16 l_ifc_ioctl_iii(l_ifc_op op, void* pv)`

Dynamic Prototype: `l_u16 l_ifc_ioctl(l_ifc_handle iii, l_ifc_op op, void* pv)`

Parameters:
 iii: Name of the interface handle to which the operation defined in op is applied
 op: Parameter used to specify the operation

pv: Pointer to a set of optional parameters for the specified operation that must be provided to the function

The following table describes the possible operations and their code values supported by the `l_ifc_ioctl` API function. The parameter list in the table shows how many parameters there are and what data type they have.

"op" Operation (Symbolic Name)	Value	"pv" Parameter List	Description
<code>L_IOCTL_READ_STATUS</code>	0x00u	None	Optional status indicators
<code>L_IOCTL_SET_BAUD_RATE</code>	0x01u	<code>l_u16*</code>	Modify baud rate
<code>L_IOCTL_SLEEP</code>	0x02u	None	Prepare device for low-power-mode entry
<code>L_IOCTL_WAKEUP</code>	0x03u	None	Restore Component state after wakeup
<code>L_IOCTL_SYNC_COUNTS</code>	0x04u	<code>l_u8*</code>	Return current number of sync field timer counts in "pv"
<code>L_IOCTL_SET_SERIAL_NUMBER</code>	0x05u	<code>l_u8*</code>	Update the pointer to the serial number
<code>L_IOCTL_GET_NAD</code>	0x06u	<code>l_u8*</code>	Return configured NAD
<code>L_IOCTL_SET_NAD</code>	0x07u	<code>l_u8*</code>	Set configured NAD
<code>L_IOCTL_GET_FRAME_PID</code>	0x08u	<code>LIN_NEW_PID*</code>	Return frame PID by frame table index
<code>L_IOCTL_SET_FRAME_PID</code>	0x09u	<code>LIN_NEW_PID*</code>	Set frame PID by frame table index
<code>L_IOCTL_SET_DNN</code>	0x0Au	<code>l_u8*</code>	Set DNN in J2602 compliance
<code>L_IOCTL_SET_FRAME_PID_BY_MESSAGE_ID</code>	0x0Bu	<code>LINS_NEW_PID_BY_MSG_ID*</code>	Set frame PID by Message ID (LIN 2.0 only)

Return Value: There is no error code value returned for the operation selected. This means that you must ensure that the values passed into the function are correct.

L_IOCTL_READ_STATUS operation

The first bit in this byte is the flag that indicates that there has been no signaling on the bus for a certain elapsed time (available when the [Bus Inactivity Timeout Detection](#) option is enabled). If the elapsed time is past a certain threshold, this flag is set. Calling this API clears all status bits after they are returned. The second bit is the flag that indicates that a Targeted Reset service request (0xB5) was received (when J2602 Compliance is enabled).

**Return Value:
(cont.)**

Symbolic Name	Value	Description
LIN_IOCTL_STS_BUS_INACTIVITY	0x0001u	No signal was detected on the bus for a certain elapsed time
LIN_IOCTL_STS_TARGET_RESET	0x0002u	Targeted Reset service request (0xB5) was received

L_IOCTL_SET_BAUD_RATE operation

Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

L_IOCTL_SLEEP operation

Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

L_IOCTL_WAKEUP operation

Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

L_IOCTL_SYNC_COUNTS operation

Returns current number of sync field timer counts for 8 bits of the synchronization field byte.

L_IOCTL_SET_SERIAL_NUMBER operation

Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

L_IOCTL_GET_NAD operation

Returns configured NAD.

L_IOCTL_SET_NAD operation

Sets configured NAD. NAD must not be 00, 7E nor 7F.

In J2602 compliance mode frame PIDs are also updated according to new NAD.

Returns 0 if operation succeeded and 1 if frame PIDs can not be updated due to NAD and frame count mismatch (see SEA J2602-1 specification 5.7.2.2 Message ID Assignment, Table 1). NAD is not updated in case of message PID update failure.

L_IOCTL_GET_FRAME_PID operation

Returns frame PIDs by frame table index.

L_IOCTL_SET_FRAME_PID operation

Sets frame PIDs by frame table index.

Uses LIN_NEW_PID structure as input parameter.

Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

L_IOCTL_SET_DNN operation

Writes DNN value to initial NAD and configured NAD, limiting its range to 0x60-0x6F.

DNN must be set at start-up if node uses hardware defined NAD.

Frame PIDs are also updated according to new NAD.

Returns 0 if operation succeeded and 1 if frame PIDs can not be updated due to NAD and frame count mismatch (see SEA J2602-1 specification 5.7.2.2 Message ID Assignment, Table 1). Initial NAD and configured NAD are updated regardless on message PID update success.

L_IOCTL_SET_FRAME_PID_BY_MESSAGE_ID operation

Sets frame PID to frame with specified Message ID (LIN 2.0 only).



Uses LINS_NEW_PID_BY_MSG_ID structure as input parameter.
Returns 0 if operation succeeded and 1 if an invalid operation parameter was passed to the function.

I_ifc_rx()

Description: The LIN Slave Component calls this API routine automatically. Therefore, this API routine must not be called by the application code. It is only listed here to show compliance with the LIN specification.

Static Prototype: void I_ifc_rx_iii(void)

Dynamic Prototype: void I_ifc_rx(I_ifc_handle iii)

Parameters: iii: Name of the interface handle

I_ifc_tx()

Description: The LIN Slave Component calls this API routine automatically. Therefore, this API routine must not be called by the application code. It is only listed here to show compliance with the LIN specification.

Static Prototype: void I_ifc_tx_iii(void)

Dynamic Prototype: void I_ifc_tx(I_ifc_handle iii)

Parameters: iii: Name of the interface handle

I_ifc_aux()

Description: The LIN Slave Component calls this API routine automatically. Therefore, this API routine must not be called by the application code. It is only listed here to show compliance with the LIN specification.

Static Prototype: void I_ifc_aux_iii(void)

Dynamic Prototype: void I_ifc_aux(I_ifc_handle iii)

Parameters: iii: Name of the interface handle

I_ifc_read_status()**Description:**

This function returns the status of the previous communication. Refer to the ISO 17987 specification for detailed information on each status information field in the LIN Slave status word.

Static Prototype: `I_u16 I_ifc_read_status_iii(void)`**Dynamic Prototype:** `I_u16 I_ifc_read_status(I_ifc_handle iii)`**Parameters:** `iii`: Name of the interface handle**Return Value:** The call returns the status word (16-bit value), as shown in the following table:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Last frame PID								0	Save configuration	Event triggered frame collision	Bus activity	Go to sleep	Over run	Successful transfer	Error in response

The status word is only set based on a frame transmitted or received by the node (except bus activity). The status word is cleared after API is called.

User-Provided Callouts

The Component does not need these callouts because it works only on Cypress chips; you can enable/disable interrupts using PSoC Creator macros.

I_sys_irq_disable()**Node Configuration Functions**

Function	Description
<code>I_id_read_configuration()</code>	Serializes the current configuration and copies it to the area (data pointer) provided by the application.
<code>I_id_set_configuration()</code>	Configures the NAD and the PIDs according to the configuration specified by input parameter.
<code>I_id_read_by_id_callout()</code>	Used when the master node transmits a read by identifier request with an identifier in the user defined area.



Id_read_configuration()

Description: This function is used to read the NAD and PID values from volatile memory. This function can be used to read the current configuration data, and then save this data into nonvolatile (flash) memory. The application should save the configuration data to flash when the "Save Configuration" bit is set in the LIN status register (returned by `l_ifc_read_status()`).

The configuration data that is read is a series of bytes. The first byte is the current NAD of the slave.

The next bytes are the current PID values for the frames that the slave responds to. The PID values are in the order in which the frames appear in the LDF or NCF file.

Dynamic Prototype: `l_u8 Id_read_configuration(l_ifc_handle iii, l_u8* const pData, l_u8* length)`

Parameters:

- iii: Name of the interface handle;
- pData: Array into which configuration data is to be read
- length: Size of configuration data in bytes. The value pointed to the length pointer parameter is set to the actual length of the configuration data.

Return Value: The function returns values listed in the following table.

Symbolic Name	Description
LD_READ_OK	Returned if the configuration data read was successful
LD_LENGTH_TOO_SHORT	Returned if the value pointed to by the length pointer parameter is less than the actual length of the configuration data

Id_set_configuration()

Description: This function is used to set the volatile NAD and PID values of the slave node. This can be used to modify the NAD and PID values at run time. This should normally only be done just after bootup or after the master requests it. Otherwise, if the slave changes its NAD or PID values, or both, the master may no longer be able to communicate with the slave.

See the `Id_read_configuration()` function for information on what the configuration data contains and how it is stored.

Dynamic Prototype: `l_u8 Id_set_configuration(l_ifc_handle iii, const l_u8* const pData, l_u16 length)`

Parameters:

- iii: Name of the interface handle
- pData: Array of configuration data which is to be applied to the slave node
- length: Size of configuration data in bytes

Return Value: The function return values are listed in the following table.

Symbolic Name	Description
LD_SET_OK	Returned if the configuration data was successfully set
LD_LENGTH_NOT_CORRECT	Returned if the value of the length parameter is not equal to the value of the configuration data of the slave node
LD_DATA_ERROR	Returned if the configuration data was not set correctly

Id_read_by_id_callout()

Description: This callout is used when the master node transmits a read by identifier request with an identifier in the user-defined area. The slave node application is called from the driver when such a request is received.

Note This function has no implementation. Implement this function with the desired functionality and override the default return value of the function which is LD_NEGATIVE_RESPONSE.

Dynamic Prototype: `I_u8 Id_read_by_id_callout (I_ifc_handle iii, I_u8 id, I_u8* frameData)`

Parameters: `iii`: Name of the interface handle

`id`: Identifier in the user defined area (32 to 63), from the read by identifier configuration request

`frameData`: Points to a data area with 5 bytes. This area is used by the application to set up the positive response.

Return Value: The function return values are listed in the following table.

Symbolic Name	Description
LD_NEGATIVE_RESPONSE	The default returns status of the API. It is always returned if you do not modify the API and reassign this to some other status.
LD_NO_RESPONSE	You can set this status manually. If set, it specifies that no response will be provided for the service.
LD_POSITIVE_RESPONSE	You can set this status manually. If set, it specifies that response will be provided for the service. The response will be pointed by the <code>frameData</code> parameter.

Transport Layer Functions

The Transport Layer is a higher-level layer of the LIN network stack. This layer allows the application to send or receive data in “message” format instead of “frame” format. Messages can be many bytes that are sent or received using multiple frames. The Transport Layer is used for configuration services, diagnostic service, or custom user-defined implementations.

API functions that send and receive Transport Layer messages have two different formats. There is a cooked format and a raw format. This Component only supports using one format of the Transport Layer API functions. The API format is chosen in the **Transport Layer** tab of the Component customizer.

Note To use the LIN Transport Layer API functions, Transport Layer use must be enabled on the **Transport Layer** tab of the LIN Slave Component customizer.

Initialization Subgroup

Function	Description
<code>Id_init()</code>	Initializes or reinitializes the raw and the cooked layers. All transport layer buffers will be initialized. If there is an ongoing diagnostic frame transporting a cooked or raw message on the bus, it will not be aborted.



Id_init()

Description: This routine initializes or reinitializes the Transport Layer of the slave node. This API must be called before using any Transport Layer API functions. It must also be called before the slave node can do any Transport Layer communication. If the API is called in the middle of an ongoing diagnostic frame transporting a cooked or raw message on the bus, the message is aborted; instead, the API waits until the message is completed.

Dynamic Prototype: void Id_init(l_ifc_handle iii)

Parameters: iii: Name of the interface handle

Raw Transport Layer API Functions Subgroup

Function	Description
Id_put_raw()	The call queues the transmission of 8 bytes of data in one frame.
Id_get_raw()	Copies the oldest received diagnostic frame data to the memory specified by input parameter.
Id_raw_tx_status()	Returns the status of the raw frame transmission function.
Id_raw_rx_status()	Returns the status of the raw frame receive function

Id_put_raw()

Description: This function is used to allow the application code to send data using the Transport Layer. It essentially copies some data from a user application array to a frame buffer array. This function is used to send one frame of a complete Transport Layer message at a time. Therefore, a multiframe Transport Layer message requires multiple calls to this API function. You should always check to see if there is a place for the frame in the buffer before calling this API.

Note In J2602 compliance mode J2602 Status is sent in first byte of each frame when:

- using 0x3C messages with NADs in the User reserved range of 0x80 - 0xFF
- using 0x3E messages with any NAD

therefor only 7 bytes of data can be sent. There is no need to place NAD as first byte of user data.

Dynamic Prototype: void Id_put_raw(l_ifc_handle iii, const l_u8* const Id_data)

Parameters: iii: Name of the interface handle

Id_data: Array of data bytes to be sent



Id_get_raw()**Description:**

This function is used to allow the application code to receive data using the Transport Layer. It essentially copies some data from a frame buffer array to a user application array. This function is used to receive one frame of a complete Transport Layer message at a time. Therefore, a multiframe Transport Layer message requires multiple calls to this API function. If the receive queue is empty, no data is copied. You should always check to see if there is a place for the frame in the buffer before calling this API.

Dynamic Prototype:

`void Id_get_raw(I_ifc_handle iii, I_u8* const Id_data)`

Parameters:

iii: Name of the interface handle

Id_data: Array to which the oldest received diagnostic frame data will be copied

Id_raw_tx_status()**Description:**

This call returns the status of the last performed frame transmission on the bus when a raw API was used.

Dynamic Prototype:

`I_u8 Id_raw_tx_status(I_ifc_handle iii)`

Parameters:

iii: Name of the interface handle

Return Value:

Symbolic Name	Description
LD_QUEUE_EMPTY	The transmit queue is empty. If previous calls to <code>Id_put_raw()</code> have been made, all frames in the queue have been transmitted.
LD_QUEUE_AVAILABLE	The transmit queue contains entries, but is not full.
LD_QUEUE_FULL	The transmit queue is full and cannot accept further frames.
LD_TRANSMIT_ERROR	LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Id_raw_rx_status()**Description:**

This call returns the status of the last performed frame reception on the bus when a raw API was used.

Note In J2602 compliance mode, when both, 0x3C and 0x3E, identifiers are used in LIN cluster, call function `I_ifc_read_status()` to identify last received PID. PID is returned in high byte of `I_u16` result. Calling `I_ifc_read_status()` clears all status bits returned in result, so take care to save the status if needed for further operations.

Dynamic Prototype:

`I_u8 Id_raw_rx_status(I_ifc_handle iii)`

Parameters:

iii: Name of the interface handle

Return Value:

Symbolic Name	Description
LD_NO_DATA	The receive queue is empty.
LD_DATA_AVAILABLE	The receive queue contains data that can be read.
LD_RECEIVE_ERROR	LIN protocol errors occurred during the transfer. Initialize and redo the transfer.



Cooked Transport Layer API Functions Subgroup

Function	Description
ld_send_message()	Packs the information specified by data and length into one or multiple diagnostic frames. The frames are transmitted to the master node with the address NAD.
ld_receive_message()	Prepares the LIN diagnostic module to receive one message and store it in the buffer pointed to by data. At the call, length specifies the maximum length allowed. When the reception has completed, length is changed to the actual length and NAD to the NAD in the message.
ld_tx_status()	Returns the status of the last made call to ld_send_message().
ld_rx_status()	Returns the status of the last made call to ld_receive_message().

ld_send_message()

Description:

This function allows the application code to send data using the Transport Layer. It is responsible for queuing up data to automatically be sent over the course of multiple SRF frames. This function is used to send a complete Transport Layer message. Therefore, a multiframe Transport Layer message requires only one call to this API function. The length value must be between 6 and 4095 bytes.

If there is a message in progress, the call returns with no action.

Note In J2602 compliance mode J2602 Status is sent in first byte of each frame instead of NAD when:

- using 0x3C messages with NADs in the User reserved range of 0x80 - 0xFF
- using 0x3E messages with any NAD.

This function also forms PCI field according to PDU structure.

Dynamic Prototype:

```
void ld_send_message(l_ifc_handle iii, l_u16 length, l_u8 nad, const l_u8* const ld_data)
```

Parameters:

iii: Name of the interface handle

length: Size of data to be sent in bytes

nad: Address of the slave node to which data is sent

ld_data : Array of data to be sent. The value of the RSID is the first byte in the data area

Side Effects:

The call is asynchronous, that is, not suspended until the message has been sent, and the buffer may not be changed by the application as long as calls to ld_tx_status() return LD_IN_PROGRESS.



Id_receive_message()**Description:**

This function allows the application code to receive data using the Transport Layer. It is responsible for receiving multiple MRF frames and copying all of the data of the message to a user application buffer array. This function is used to receive a complete Transport Layer message. Therefore, a multiframe Transport Layer message requires only one call to this API function. The length value must be between 6 and 4095 bytes.

Note In J2602 compliance mode, when both, 0x3C and 0x3E, identifiers are used in LIN cluster, call function `l_ifc_read_status()` to identify last received PID. PID is returned in high byte of `l_u16` result. Calling `l_ifc_read_status()` clears all status bits returned in result, so take care to save the status if needed for further operations.

When using 0x3C messages with NADs in the User reserved range of 0x80 – 0xFF or using 0x3E messages with any NAD, this function also requires presense of PCI field according to PDU structure to get type of frame and length of message.

Dynamic Prototype:

```
void Id_receive_message(l_ifc_handle iii, l_u16* const length, l_u8* const nad,  
l_u8* const ld_data)
```

Parameters:

iii: Name of the interface handle

length: Size of data to be received in bytes

nad: Address of the slave node from which data is received

ld_ata: Array of data to be received. The value of the SID is the first byte in the data area.

Side Effects:

The call is asynchronous, that is, not suspended until the message has been received, and the buffer may not be changed by the application as long as calls to `Id_tx_status()` return `LD_IN_PROGRESS`.



ld_tx_status()

Description: This function returns the status of the last call made to `ld_send_message()` and the last Transport Layer data transmission on the bus.

Dynamic Prototype: `I_u8 ld_tx_status(I_ifc_handle iii)`

Parameters: `iii`: Name of the interface handle.

Return Value: The following values can be returned.

Symbolic Name	Description
LD_IN_PROGRESS	The transmission is not yet completed.
LD_COMPLETED	The transmission has completed successfully (and you can issue a new <code>ld_send_message</code> call()). This value is also returned after initialization of the transport layer.
LD_FAILED	The transmission ended in an error. The data was only partially sent. The transport layer must be reinitialized before processing further messages. To find out why a transmission has failed, check the status management function <code>I_read_status()</code> .
LD_N_AS_TIMEOUT	The transmission failed because of an N_As timeout, and current message transmission will be aborted. See Section 7.6.1 of the ISO 17987 specification.

Note

If the transmission failed, the error status (e.g., `LD_FAILED`) could be read just once by this function. The next read will return `LD_COMPLETED` status.

This is because the LIN Component automatically initializes the status to `LD_COMPLETED` during transport layer re-initialization, due to an error in the message transmission. This is a result of the automated recovery mechanism embedded in the LIN Slave Component.



ld_rx_status()

Description: This function returns the status of the last call made to `ld_receive_message()` and the last Transport Layer data reception on the bus.

Dynamic Prototype: `I_u8 ld_rx_status(I_ifc_handle iii)`

Parameters: `iii`: Name of the interface handle

Return Value: The following values can be returned:

Symbolic Name	Description
<code>LD_IN_PROGRESS</code>	The reception is not yet completed.
<code>LD_COMPLETED</code>	The reception has completed successfully and all information (length, NAD, data) is available. You can also issue a new <code>ld_receive_message()</code> call. This value is also returned after initialization of the transport layer.
<code>LD_FAILED</code>	The reception ended in an error. The data was only partially received and should not be trusted. Initialize before processing further transport layer messages. To find out why a reception has failed, check the status management function <code>I_read_status()</code> .
<code>LD_N_CR_TIMEOUT</code>	The reception failed because of an <code>N_Cr</code> timeout, and current message reception will be aborted. See Section 7.6.1 of the ISO 17987 specification.
<code>LD_WRONG_SN</code>	The reception failed because of an unexpected sequence number.

Note

If the reception failed, the error status (e.g., `LD_WRONG_SN`) could be read just once by this function. The next read will return `LD_COMPLETED` status.

This is because the LIN Component automatically initializes the status to `LD_COMPLETED` during transport layer re-initialization, due to an error in the received message. This is a result of the automated recovery mechanism embedded in the LIN Slave Component.

Non-LIN-Specified API

Function	Description
<code>LIN_Start()</code>	Starts the Component operation.
<code>LIN_Stop()</code>	Stops the Component operation.
<code>LIN_j2602Status_Reset()</code>	Clears all unreported states in the <code>ERR</code> bits of J2602 Status byte and sets Reset state.

LIN_Start()

Description: Starts the Component operation. This function is not required.

Dynamic Prototype: `I_bool LIN_Start()`

Return Value: Zero: The initialization succeeded.

Nonzero: The initialization failed.



LIN_Stop()

Description: Stops the Component operation. This function is not required.

Dynamic Prototype: `I_bool LIN_Stop()`

LIN_j2602Status_Reset()

Description: Clears all unreported states in the ERR bits of J2602 Status byte sets Reset state. This function is not required if Service B5 is handled by component.

Dynamic Prototype: `none`

Macro Callbacks

Macro callbacks allow users to execute code from the API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator Help and *Component Author Guide* for the more details.

In order to add code to the macro callback present in the Component's generated source files, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will “uncomment” the function call from the Component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.
- Write the function implementation (in any user file).

Callback Function ^[1]	Associated Macro	Description
<code>LIN_UART_ISR_EntryCallback</code>	<code>LIN_UART_ISR_ENTRY_CALLBACK</code>	Used at the beginning of the <code>LIN_UART_ISR()</code> interrupt handler to perform additional application-specific actions.
<code>LIN_UART_ISR_ExitCallback</code>	<code>LIN_UART_ISR_EXIT_CALLBACK</code>	Used at the end of the <code>LIN_UART_ISR()</code> interrupt handler to perform additional application-specific actions.
<code>I_ifc_rx_LIN_Callback</code>	<code>L_IFC_RX_LIN_CALLBACK</code>	Used in the <code>I_ifc_rx_LIN()</code> function to perform additional application-specific actions.
<code>I_ifc_aux_LIN_Callback</code>	<code>L_IFC_AUX_LIN_CALLBACK</code>	Used in the <code>I_ifc_aux_LIN()</code> function to perform additional application-specific actions.

¹ The callback function name is formed by Component function name optionally appended by short explanation and “Callback” suffix.

Callback Function ^[1]	Associated Macro	Description
Id_read_by_id_callout_LIN_Callback	LD_READ_BY_ID_CALLOUT_LIN_CALLBACK	Used in the Id_read_by_id_callout_LIN() function to perform additional application-specific actions.

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For Component-specific examples, open the dialog from the Component Catalog or an instance of the Component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the Component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator Components
- specific deviations – deviations that are applicable only for this Component

This section provides information on Component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The LIN Slave Component is MISRA compliant, except for following specific deviations:

MISRA-C:2004 Rule	Rule Class (Required/Advisory)	Rule Description	Description of Deviation(s)
1.1	R	This rule states that code shall conform to C ISO/IEC 9899:1990 standard.	Nesting of control structures (statements) exceeds 15 - program does not conform strictly to ISO:C90. In practice, most compilers will support a much more liberal nesting limit and therefore this limit may only be relevant when strict conformance is required. By comparison, ISO:C99 specifies a limit of 127 "nesting levels of blocks.



MISRA-C:2004 Rule	Rule Class (Required/Advisory)	Rule Description	Description of Deviation(s)
8.7	R	Objects shall be defined at block scope if they are only accessed from within a single function. That is, minimize the scope of objects and variables.	The reason of this violation is that in some configurations following internal variables are used only in one function: LIN_LinSlaveConfig, LIN_prevPci, messagelDTable
8.8	R	An external object or function shall be declared in one and only one file.	The interrupt service routine (ISR) for COM void ComExt_UART_ISR (void); is defined only once in the file ComExt_INT.c, which includes header ComExt.h with function declaration. No possible issues expected.
11.3	A	Cast between a pointer to volatile object and an integral type.	Casting performed when accessing SCB internal registers.
11.4	A	A cast should not be performed between a pointer to object type and a different pointer to object type.	Section 7.2.5.4 of LIN 2.2 specification defines l_ifc_ioctl() with following prototype – l_u16 l_ifc_ioctl (l_ifc_handle iii, l_ioctl_op op, void* pv). Depending on the operation the “pv” parameter may be converted to pointer to unsigned char or to unsigned short (l_u16).
12.1	A	Limited dependence should be placed on C's operator precedence rules in expressions.	Extra parentheses recommended to be used to emphasise order of operations for LIN_ET_FRAMES_FLAGS_SIZE define evaluation.
12.4	R	Right hand operand of '==' or '==' is an expression with possible side effects.	Expression operates with volatile variable. It is safe as these variables are accessed only from interrupt routine.
13.7	R	Boolean operations whose results are invariant shall not be permitted.	Depending on the Component setup there may be condition checks whose results are invariant. For example when ld_read_by_id_callout() is used it always returns LD_NEGATIVE_RESPONSE unless the user will override it. But Component performs condition check for all of three possible return values in its source code.
14.1	R	There shall be no unreachable code. This refers to code which cannot, under any circumstances, be reached.	This comes in pair with 13.7. Depending on the Component setup there may be condition checks whose results are invariant. This results in unreachable code.



MISRA-C:2004 Rule	Rule Class (Required/Advisory)	Rule Description	Description of Deviation(s)
14.2	R	The statement has no side-effect – it can be removed.	There are conditionally included parts of code and user-defined callback functions. Function parameters may be unused if such code is not enabled. “(void) id;” statements are used to prevent unused variables warnings.
14.7	R	A function shall have a single point of exit at the end of the function.	The L_ifc_rx() function has a complex conditional structure and two more ‘return’ paths are added to return immediately after receiving of BREAK sequence or if spurious interrupt occurs.
15.5	R	This ‘switch’ statement contains only a single path - it is redundant.	This ‘switch’ is generated in customizer. Number of its paths depends on quantity of signals or frames defined by user.
16.7	A	A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.	Section 7.2.5.4 of LIN 2.2 specification defines L_ifc_ioctl() with following prototype – L_u16 L_ifc_ioctl (L_ifc_handle iii, L_ioctl_op op, void* pv). In some cases, depending on configuration the “pv” parameter of may not be modified.
17.4	R	Array indexing shall be the only allowed form of pointer arithmetic. This still bans the incrementing of a pointer that was not declared as an array.	To conform to the LIN 2.2 specification the Component defines several API functions that use pointers as parameters. These pointers are used to define arrays of data and array indexing is used to access the data.
19.7	A	A function shall be used in preference to a function-like macro	The following macro is used to increase performance: LIN_SWAP_U8_TO_U16(); LIN_ABS().
19.11	R	All macro identifiers in preprocessor directives shall be defined before use, except in #ifdef and #ifndef preprocessor directives and the defined() operator.	The macro ‘LIN_1_NCS_0xB5_SEL’ used in '#if' expression in L_ifc_ioctl_LIN_1() function is not defined in case if Automatic Configuration Request Handling option is unchecked.



API Memory Usage

The Component memory usage varies significantly, depending on the compiler, device, number of APIs used and Component configuration. The following table provides the memory usage for all APIs available in the given Component configuration.

The measurements have been done with associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 4 (GCC)	
	Flash Bytes	SRAM Bytes
P4_SCB_LIN_Slave_Example project	7840	584

Functional Description

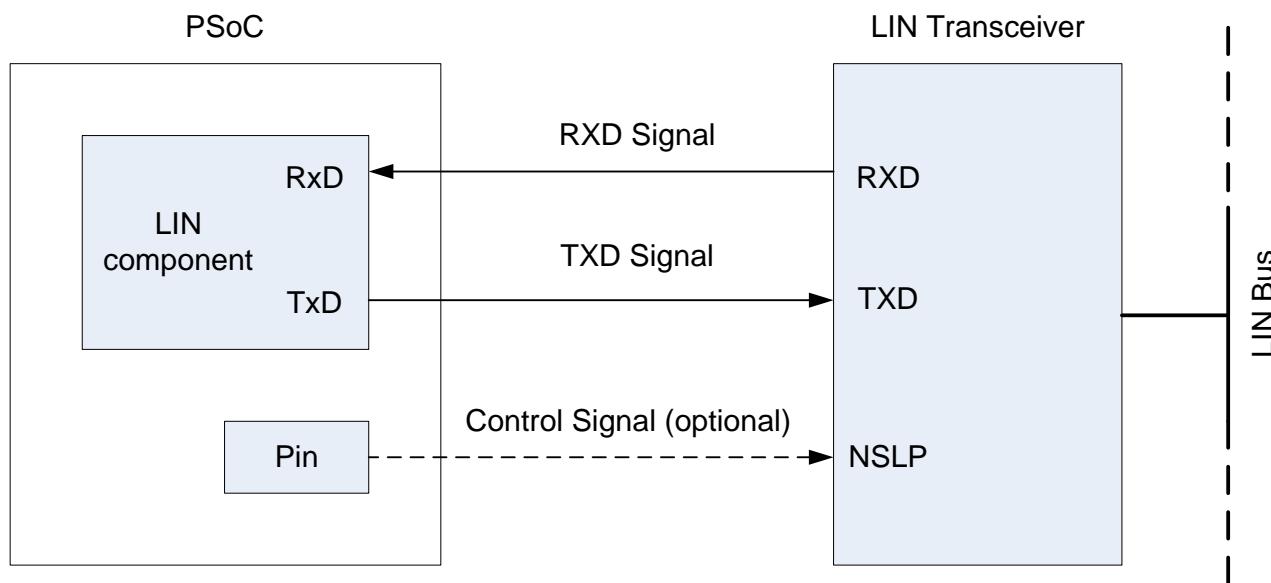
PSoC and LIN Bus Hardware Interface

You need a LIN physical layer transceiver device when the PSoC LIN slave node is connected directly to a LIN bus. In this case, the TxD pin of the LIN Component connects to the TXD pin of the transceiver, and the RxD pin connects to the RXD pin of the transceiver. The LIN transceiver device is required because the PSoC's electrical signal levels are not compatible with the electrical signals on the LIN bus.

Some LIN transceiver devices also have an "enable" or "sleep" input signal that is used to control the operational state of the device. The LIN Component does not provide this control signal. Instead, use a pin used to output the desired signal to the LIN transceiver device if this signal is needed.

Note After the Component goes into Sleep mode, the SCB is disconnected from the TxD pin and this pin goes into high-impedance state. It is a responsibility of the user's code to avoid generation of low-level pulse. The simplest way is to disable the external LIN transceiver before entering Sleep mode. While disabled, the transceiver can still transfer wake-up pulses from the LIN bus to the RxD pin, so a wake-up event will not be missed.



Figure 10. Hardware Interface between PSoC and LIN Bus

SysTick Timer Usage

For PSoC 4 devices, this Component uses the SysTick timer to monitor LIN bus inactivity and measure sleep delays.

The SysTick timer is part of the `cy_boot` Component. All manipulation with the SysTick timer can be done by using `cy_boot` APIs. The SysTick timer is configured by `cy_boot` and enabled by default. If the SysTick timer is disabled before LIN initialization, then the `LIN_TimerEnableInt()` function configures and enables it. For more information about `cy_boot`, refer to the *System Reference Guide*.

The Bus Inactivity Timer is based on the Cortex-M0 CPU's SysTick timer. It is configured by default at Reset time to interrupt period 1 millisecond. The LIN Component uses one callback handler (out of five) from the SysTick timer. Two LIN instances in the project use two callback handlers from the SysTick timer.

The CapSense Component can share SysTick timer usage. It also configures the SysTick timer to 1 millisecond period.

A LIN user may use the SysTick interrupt for custom purposes and change the interrupt period. In that case, the LIN inactivity function will get overridden and bus inactivity will not be detected. The LIN Component relies on a 1 millisecond interrupt period.

If J2602 compliance is enabled, SysTick interrupt is also used to detect errors in the SYNC field. This functionality is active if automatic baud rate synchronization is disabled. Changing the SysTick interrupt period will affect this functionality. Use automatic baud rate synchronization to disable SYNC field error detection.



If the SysTick interrupt period remains 1 milisecond, set the register callback for the SysTick interrupt by using the CySysTickSetCallback() function.

Response Pending

For PSoC 4 devices, the Component supports the response pending frames (UDSonLIN based diagnostic service). See ISO 14229-2 and ISO 17987-2 for more details.

The response pending frame is defined as a SingleFrame negative response with NRC 78₁₆:

Table 2. Response Pending Frame Format

NAD	PCI	RSID	D1	D2	D3	D4	D5
NAD	03 ₁₆	7F ₁₆	SID	78 ₁₆	FF ₁₆	FF ₁₆	FF ₁₆

The response pending frame could be send by using Id_put_raw() or Id_send_message() Transport Layer APIs. For Id_put_raw(), the API sets the Id_data parameter to SingleFrame defined in [Table 2](#). For Id_send_message(), the API sets the Id_data parameter to RSID, D1, D2 specified in [Table 2](#) and length parameter equal to 3.

Note Whenever a response pending frame is used a final positive or negative response is mandatory for this request.

Resources

The LIN Component (for PSoC 4) is based on an SCB fixed block. The Component uses the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
P4_SCB_LIN_Slave_Example project	–	–	–	–	–	1+1 (SysTick callback handler)
P4_SCB_LIN_Slave_Multi_Instance Example project	–	–	–	–	–	2+2 (SysTick callback handlers)

DC and AC Electrical Characteristics

For information about DC and AC Electrical Characteristics refer to the “LIN Physical Layer Specification” chapter of the *LIN 2.2 Specification*.

Specifications are valid for $-40^{\circ}\text{C} \leq T_A \leq 85^{\circ}\text{C}$ and $T_J \leq 100^{\circ}\text{C}$, except where noted.

Specifications are valid for 1.71 V to 5.5 V, except where noted.

Note Final characterization data for PSoC Analog Coprocessor device is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.

DC Characteristics

Fixed UART DC Specifications

Parameter	Description	Min	Typ	Max	Units
I _{UART1}	Block current consumption at 100 Kbit/s	—	—	9	µA
I _{UART1}	Block current consumption at 100 Kbit/s for PSoC 4100 / PSoC 4200, PSoC 4100 BLE / PSoC 4200 BLE, PSoC 4100M / PSoC 4200M, PSoC 4200L, PSoC 4000S, and PSoC 4100S	—	—	55	µA

Performance Characteristics

Symbol	Description	Conditions	Min	Typ	Max	Units
F _{BAUD}	Baud Rate	Clock = 66 MHz	1	-	20	kbaud
		Clock = 48 MHz	1	-	20	kbaud
		Clock = 24 MHz	1	-	20	kbaud
		Clock = 12 MHz	1	-	10	kbaud
		Clock = 6 MHz	1	-	5	kbaud
		Clock = 3 MHz	1	-	2.5	kbaud
N _{ISR}	ISR Length		-	-	729	CPU Cycles
t _{ISR LAT}	ISR Latency		-	-	1 / (F _{BAUD})	s

F_{BAUD} parameter is limited by HFCLK clock for PSoC 4 devices, only when automatic baud rate synchronization is enabled (for clock source with a frequency deviation of ± 1.5 percent or more).

The above F_{BAUD} values are calculated from the following formula:

$$F_{BAUD\ MAX} < F_{BUS_CLK} / (16 * DIVIDER_VALUE)$$



Where DIVIDER_VALUE = 75. This conservative value provides frequency tuning with granularity of $1/75 = 1,33\%$. As LIN Spec. defines deviation after synchronization $< +/-2\%$, DIVIDER_VALUE can be reduced to 25 (4%) for clock source with low short-term deviation.

Maximal ISR Latency is 1 bit-time because transmitter must be disabled at the edge of new symbol when previously sent symbol received back as erroneous.

Component Errata

Cypress ID	Version	Problem	Workaround
335998	All	LIN slave reports incorrect message ID when LIN2.0 specification compatibility is selected. LSB and MSB of the message ID are swapped in slave response.	<p>In the LIN.c file, find the following code:</p> <pre>/*D1= Message ID LSB */ LINS_srfBuffer[LINS_PDU_D1_IDX] = HI8(messageIdTable[tempStatus]);</pre> <pre>/* D2 = Message ID MSB */ LINS_srfBuffer[LINS_PDU_D2_IDX] = LO8(messageIdTable[tempStatus]);</pre> <p>Swap the HI8() and LO8() functions, so that LSB is LO8 and MSB is HI8.</p> <p>IMPORTANT: The LIN.c file is a generated file, and it is overwritten each time you build the application. Therefore, you must re-edit the LIN.c file every time you build or generate code.</p>

Component Changes

This section lists the major changes in the Component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
5.0.b	Updated datasheet.	Added rule 8.8 to MISRA Compliance table.
5.0.a	Edited datasheet.	Added Errata item, Cypress ID 335998, to document an issue with Message ID in response to a read by ID service.
5.0	Added ISO 17987 and SAE J2602 compliance support. Added response pending frames support. Removed support for PSoC 3 and PSoC 5 devices. Added MISRA violation 14.2.	New protocols support added.
4.0.a	Edited datasheet.	Updated certification statement for PSoC 4 devices.
4.0	Updated LIN_ProcessMrf() API implementation.	Fixed Component Errata with Cypress ID 245706.
	Updated ClrRxBuffer(), ClrTxBuffer() and Id_tx_status_LIN(), Id_tx_status_LIN() APIs implementation.	Fixed Component Errata with Cypress ID 250810.
3.40.d	Edited datasheet.	Added Errata item, Cypress ID 244944, to document using diagnostic frames. Added Errata item, Cypress ID 250810, to document behavior when erroneous TL packets reseived. Updated Id_rx_status() API descriptoin, to document Component behavior when receiving a broken packet. Added clarification about partial compliance with SAE J2602-1 specification. Added SysTick timer usage section. Added final characterization data for the PSoC 4100S device.
3.40.c	Edited datasheet.	Added Errata item, Cypress ID 245706, to document issue with the cooked transport layer. Added final characterization data for PSoC 4000S device.
3.40.b	Edited datasheet.	Added a note that certification is pending for the 3.40 version of this Component.
3.40.a	Fixed the following issues in the I_ifc_rx_LIN() function related to the Transport layer:	Customer Request.



Version	Description of Changes	Reason for Changes / Impact
	<ul style="list-style-type: none"> - An incorrect frame length value was placed in the corresponding field of the first frame of the Segmented Response message; - The actual NAD address was not updated in the TL frame header each time when the frame was constructed. - Erroneous length check was applied to the last frame of the Segmented Response message. 	
	Edited datasheet.	<p>Final characterization data for PSoC 4000S, PSoC 4100S and PSoC Analog Coprocessor devices is not available at this time. Once the data is available, the Component datasheet will be updated on the Cypress web site.</p> <p>Updated MISRA section.</p>
3.40	Updated versions of the embedded SCB Component to the most current version.	Out of date Components may contain defects or incompatibilities.
3.30	Added PSoC 4200L device support, and updated characterization data.	New device support.
3.20.b	Datasheet update.	<p>Added Macro Callbacks section.</p> <p>Added certification statement for PSoC 4 devices, as well as a statement to explain that this a prototype Component for PSoC 3 and PSoC 5LP.</p>
3.20.a	Added a note to the Bus Inactivity Timeout Detection parameter section.	Clarified the minimal timeout value and compatibility with LIN 1.3.
3.20	Added Errata section	Component validation
	Added LIN_Dynamic Component section.	To explain the use of the LIN_Dynamic Component.
	Added recommendations about Sleep mode entering procedure concerned to control of external transceiver	Validation report
	Added 11.3,12.4,14.7,15.5 MISRA violations	MISRA related change.
	Added support for SCB based LIN Component	Component support for PSoC 4200 device family.
	Break Detection Threshold setting options needs to be changed based on the hardware implementation	The break width options available for SCB are limited (only whole no. values). This automatically has to update the drop down options based on hardware implementation (UDBs for PSoC 3/ PSoC 5LP; SCB for PSoC 4).
	Any integer field in an imported NCF/LDF file should be imported properly if it is an integer value in decimal float format (example: 10.0).	Customer Request.

Version	Description of Changes	Reason for Changes / Impact
	Allow multiple instances of LIN Component (v2.x and v1.3) in a single project based on hardware availability.	Customer Request. Component APIs might get affected based on the number of Component instance in the project.
	Added Hover window settings as Auto Baud rate sync, LIN bus Baud rate and slave information (Supplier ID & Function ID)	Based on review comments
	Edited the datasheet.	Updated DC and AC Electrical Characteristics section with PSoC 4100M/PSoC 4200M data.
1.30.b	Added Component Errata section applicable to v1.30 only.	Document that the Component was changed, but there is no impact to designs.
1.30.a	Minor datasheet edits.	
1.30	Support of PSoC 5 family devices was removed from the Component.	
	Following variables were declared as "static": LIN_Slave_FindPidIndex(); LIN_Slave_EndFrame(); LIN_Slave_SetAssociatedFlags(); LIN_Slave_GetEtFlagValue(); LIN_Slave_ClearEtFlagValue(); LIN_Slave_ProcessMrf(); LIN_Slave_LinProductId(); LIN_Slave_MessageId().	MISRA related change. These functions are designated only for internal Component's usage.
	The number of 45 global variables were declared as "static"	MISRA related change. These variables are designated only for internal Component's usage.
	The MISRA Compliance section was updated.	The Component was verified for MISRA compliance.
	The API Memory Usage section was updated	The new API memory usage numbers were inserted.
	Added MISRA Compliance section.	The Component was not verified for MISRA compliance.
1.10	Updated LIN Slave with the latest version of the Clock and Interrupt Components	
	Updated Component characterization data.	
	Added PSoC 5LP support.	
	Added all Component APIs with the CYREENTRANT keyword when they are included in the .cyre file.	Not all APIs are truly reentrant. Comments in the Component API source files indicate which functions are candidates.



Version	Description of Changes	Reason for Changes / Impact
		This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections.
	Description of 0xB5 service was modified to insert more clarity on the service usage depending on Component configuration.	
1.0.a	Minor datasheet edits and updates	

© Cypress Semiconductor Corporation (an Infineon company), 2019-2021. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical Components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical Component is any Component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.

