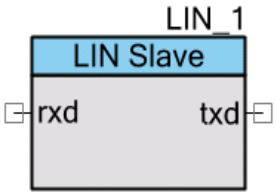
LIN Slave
1.10

Features



- Full LIN 2.1 or 2.0 Slave Node implementation
- Supports compliance with SAE J2602-1 specification
- Automatic baud rate synchronization
- Fully implements a Diagnostic Class I Slave Node
- Full transport layer support
- Automatic detection of bus inactivity
- Full error detection
- Automatic configuration services handling
- Customizer for fast and easy configuration
- Import of **.ncf*/**.ldf* files and **.ncf* file export
- Editor for **.ncf*/**.ldf* files with syntax checking

General Description

The LIN Slave component implements a LIN 2.1 slave node on PSoC 3 and PSoC 5 devices. Options for LIN 2.0 or SAE J2602-1 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 LIN 2.1 Specification.

This component provides a good combination of flexibility and ease of use. A customizer for the component is provided that allows you to easily configure all parameters of the LIN Slave.

Definitions

Many of the definitions given in this datasheet are from the LIN 2.1 specification. In these cases, refer to the specified section of the LIN 2.1 specification for a proper understanding of the term.

Input/Output Connections

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

TXD – Output

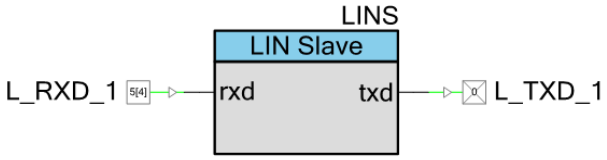
This is a digital output terminal. This terminal's signal is the data that this LIN node sends onto the LIN bus.

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.

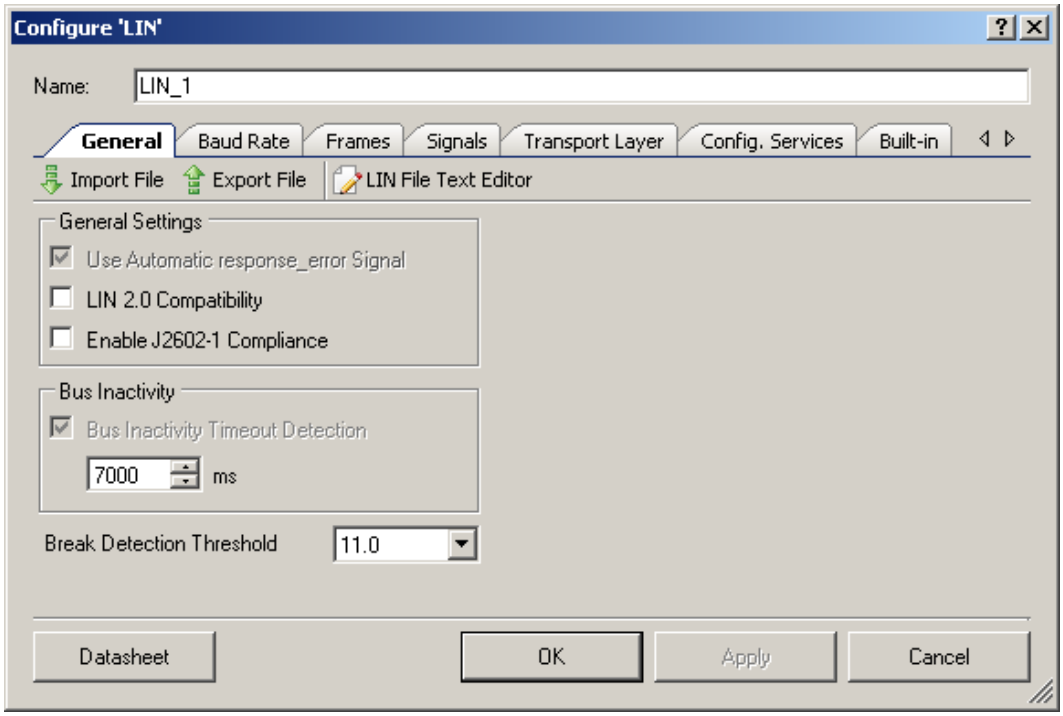
Schematic Macro Information

By default, the PSoC Creator Component Catalog contains a schematic macro for the LIN component. This macro contains already connected and configured pin components. The schematic macro with the default component configuration is shown below.



Component Parameters

Drag a LIN Slave component onto your design and double click it to open the **Configure LIN** dialog.



General Tab

Use Automatic response_error Signal

This check box on the tab sets the automatic error signal selection. This box is always selected, so a 1-bit signal is automatically added in the **Signals** tab of the customizer. 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 LIN 2.1 specification.

LIN 2.0 Compatibility

This option selects whether this component is compatible with the LIN 2.0 specification. The status of this check box affects other areas of the customizer.

Enable J2602-1 Compliance

The SAE J2602-1 specification is parallel to the LIN 2.x specifications. It adds restrictions to the LIN 2.x requirements. However, there are also a few extra features that are supported by this component that make it J2602-1 compliant. The status of this check box affects other areas of the customizer.

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 L_IOCTL_READ_STATUS operation of the l_ifc_ioctl() function. See the function description for more information.

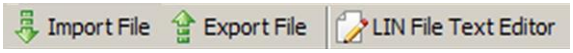
Break Detection Threshold

This option configures slave node break detection threshold. Default value is 11 dominant local slave bit times. See section 2.3.1.1 of the LIN 2.1 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.

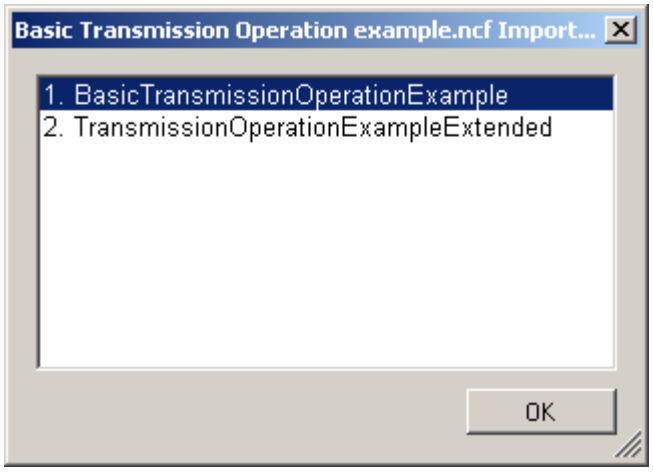
Figure 1. General Toolbar



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 2. Choose one of the available node descriptions to import.

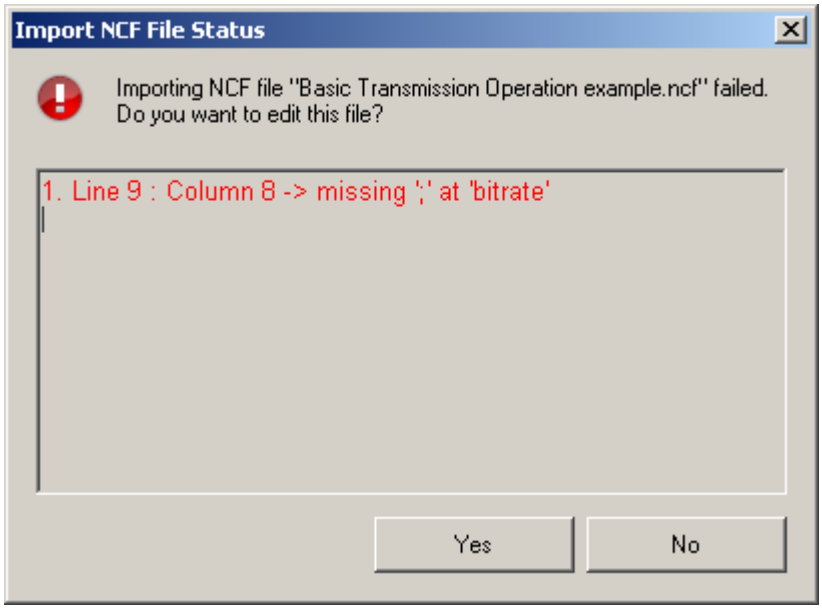
Figure 2. List of Available Nodes of NCF File to Import



The syntax for **.ncf* and **.ldf* files is verified according to the LIN Node Capability Language Specification (Revision 2.1) and to the LIN Configuration Language Specification (Revision 2.1), respectively.

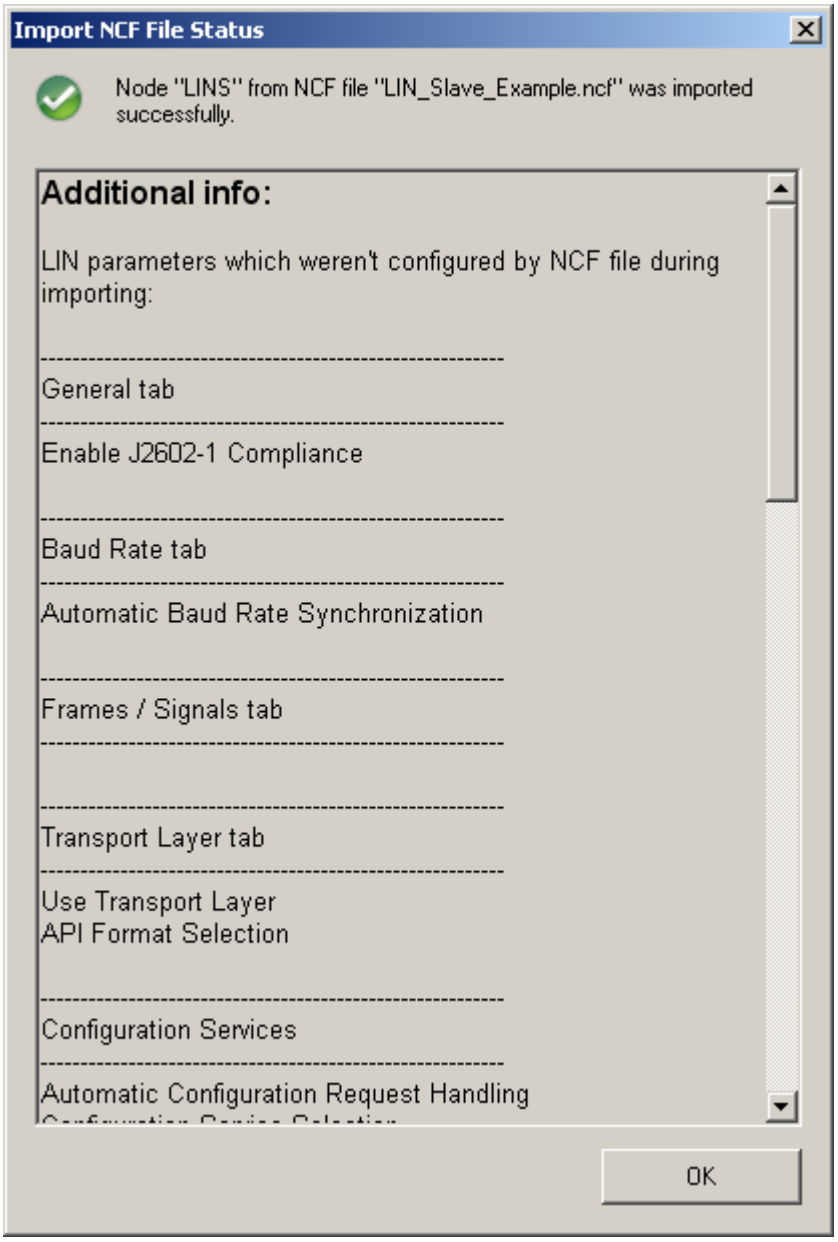
If the imported file contains errors, a dialog window similar to Figure 3 is displayed. There are two options in this case: edit the imported file to correct the errors using LIN Enhanced Editor Tool (see LIN File Text Editor for more information) or cancel the import by clicking the **No** button.

Figure 3. NCF File Import Failed



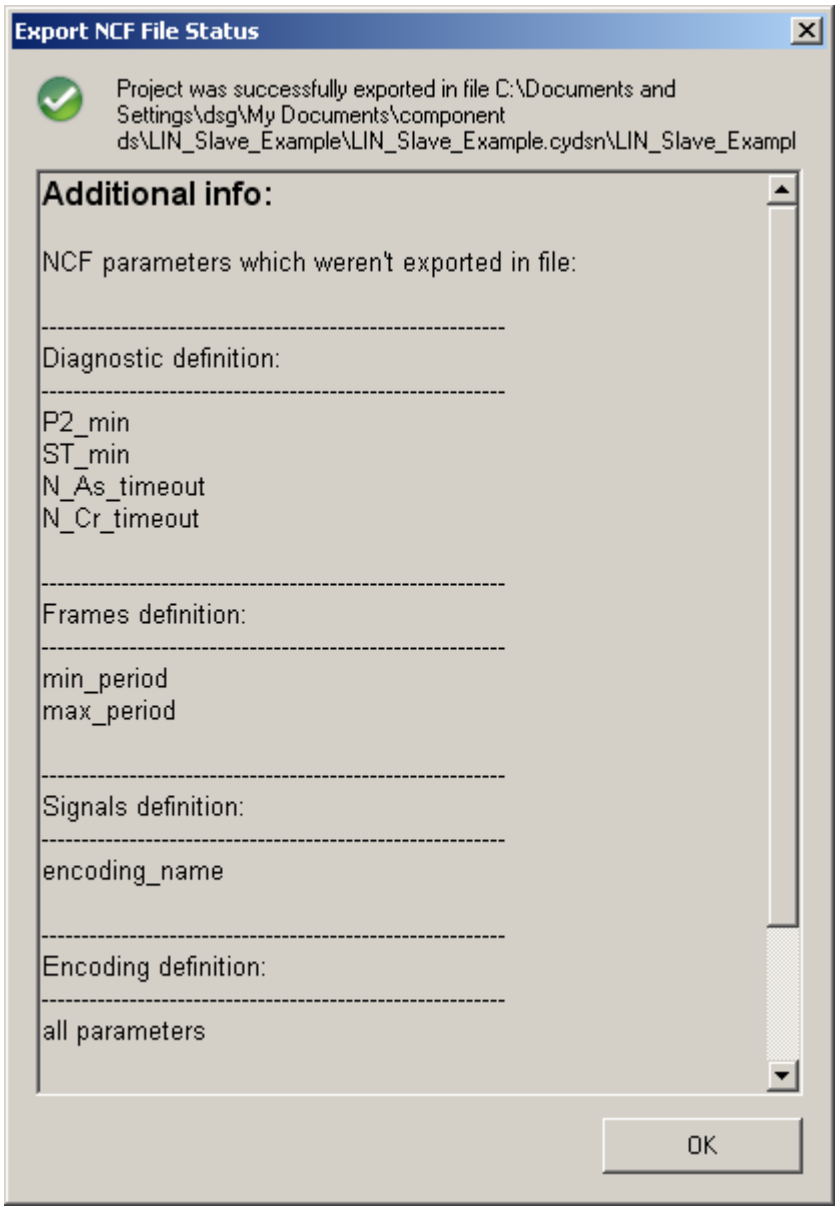
After the node to import is chosen and the import to the customizer is completed, a dialog box that describes the importing results is displayed (see Figure 4). The importing results contain the LIN Slave component parameters that were not affected during import.

Figure 4. NCF File Import Information



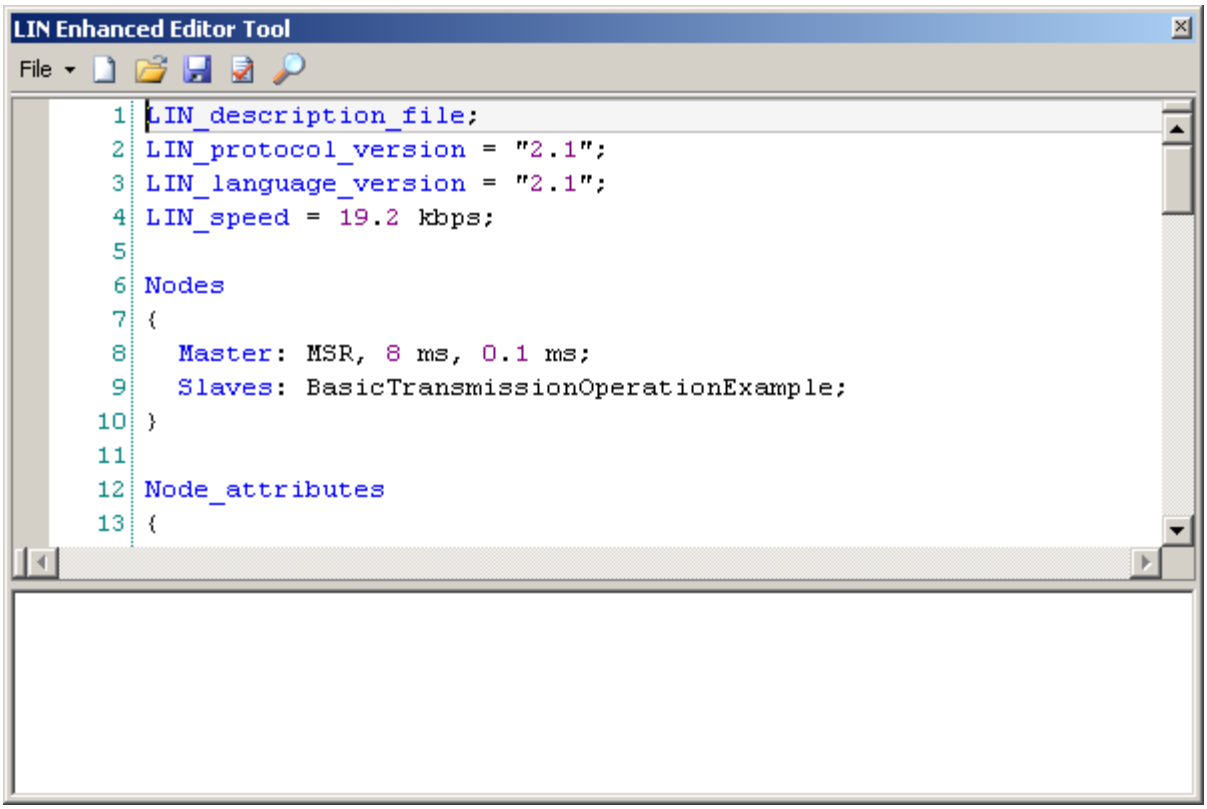
Export File: This tool enables you to save information about the component configuration into a Node Capability File (NCF).

Figure 5. NCF File Export Information



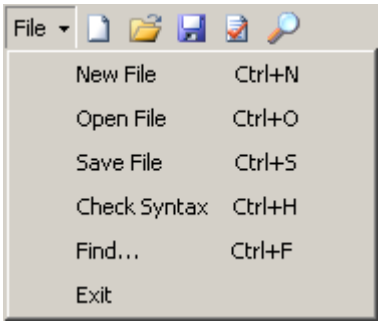
LIN File Text Editor: This tool is used to cheat, edit, and verify the syntax of the NCF/LDF file. The syntax for **.ncf* files is verified according to the *LIN Node Capability Language Specification* (Revision 2.1). The syntax for **.ldf* files is verified according to the *LIN Configuration Language Specification* (Revision 2.1).

Figure 6. LIN File Text Editor Tool



There is a toolbar at the top of the **LIN Enhanced Editor Tool** window (see Figure 7).

Figure 7. LIN File Text Editor Toolbar



New File: Creates a new file of the selected LIN file type.

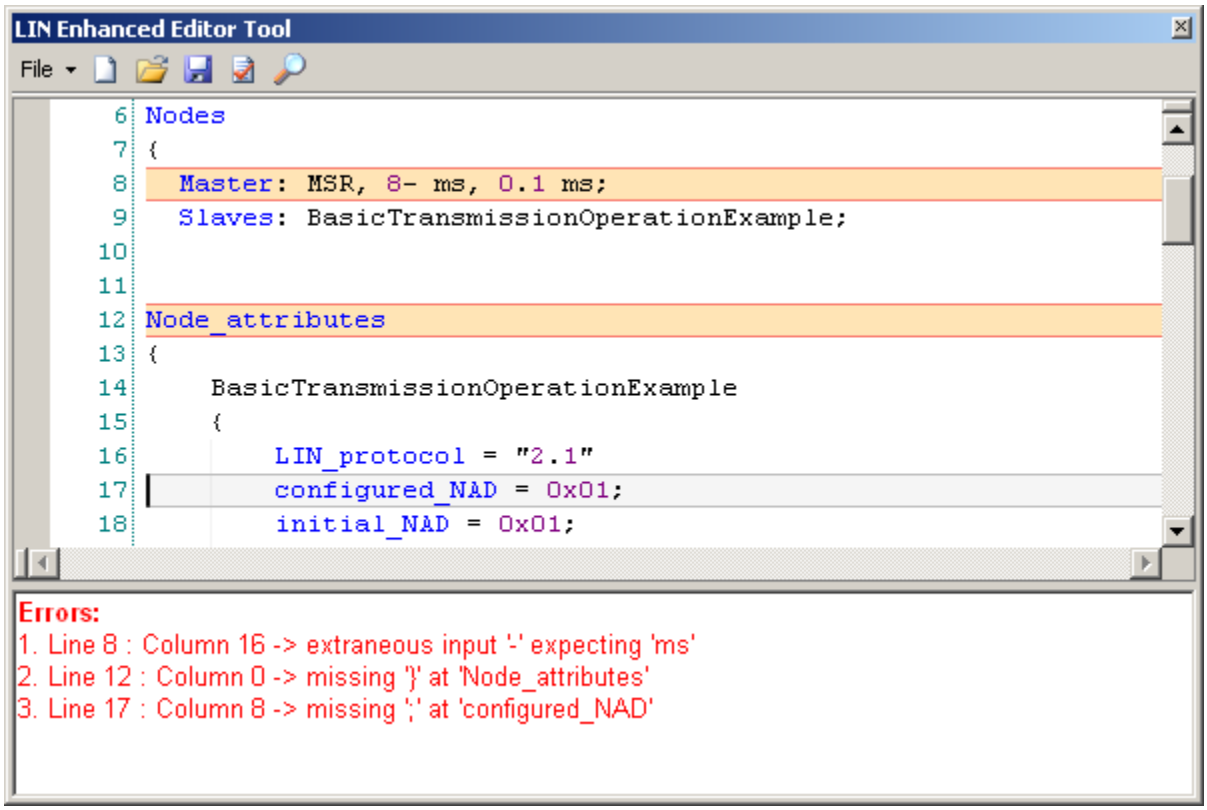
Open File: Opens the specified existing LIN file.

Save File: Saves the created LIN file to the specified location.

Check Syntax: This control allows you to check whether an **.ncf*/**.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 8). The code lines containing errors are highlighted in red.

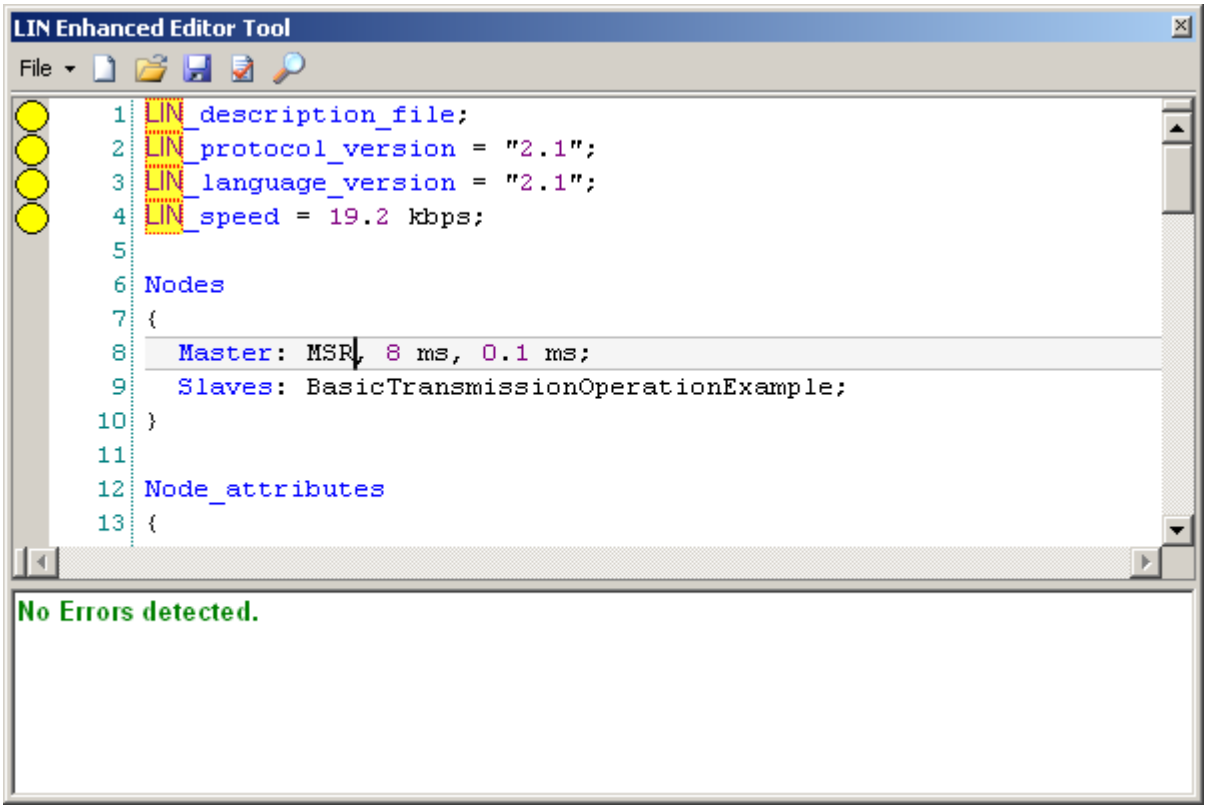
Double-clicking the error line in the output area navigates to the line containing an error in file.

Figure 8. LIN File Syntax Check



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 9. The **Clear** button removes all highlighted tokens.

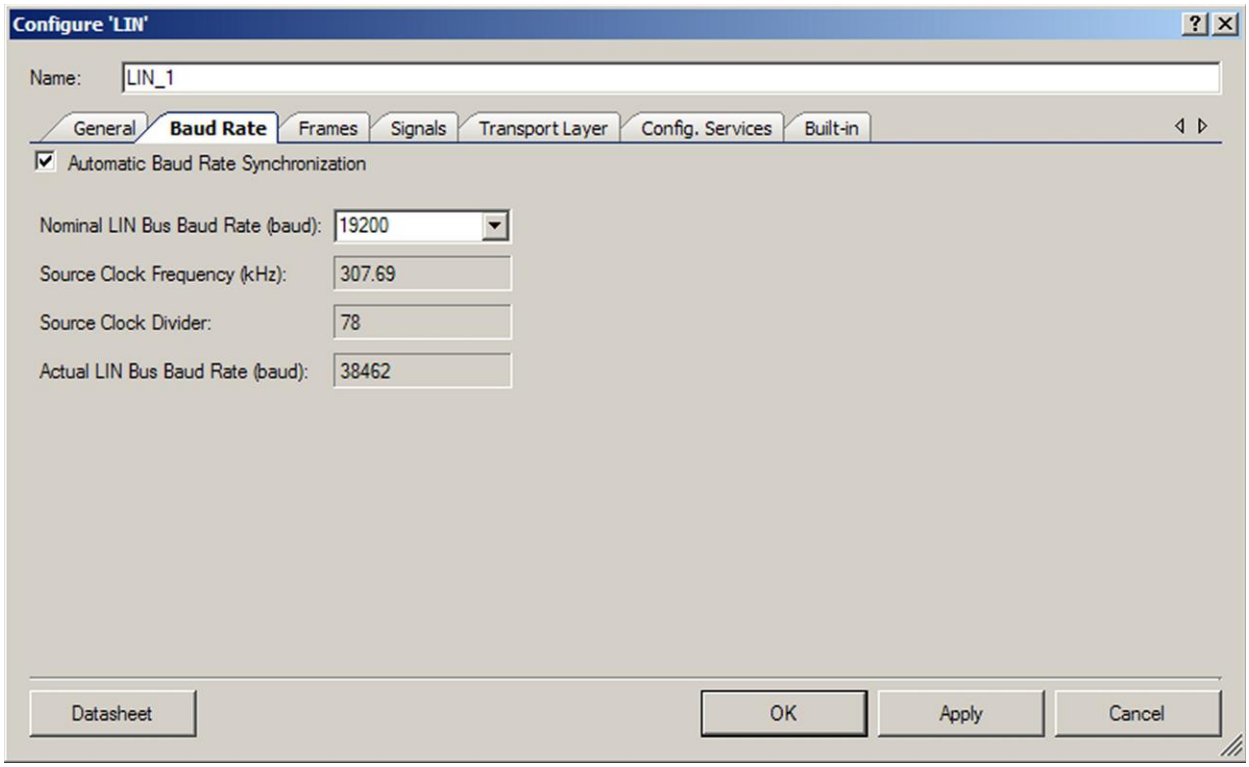
Figure 9. LIN File Finding Results



All tools are also available in the **File** menu of the LIN Enhanced Editor Tool (see Figure 6) and through the appropriate toolbar commands.

Baud Rate Tab

Figure 10. Configure LIN Dialog, 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 LIN 2.1 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 BusClk 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, you can type in any value between 1000 and 20000 in the combo box. 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 8, 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 BusClk.

Actual LIN Bus Baud Rate

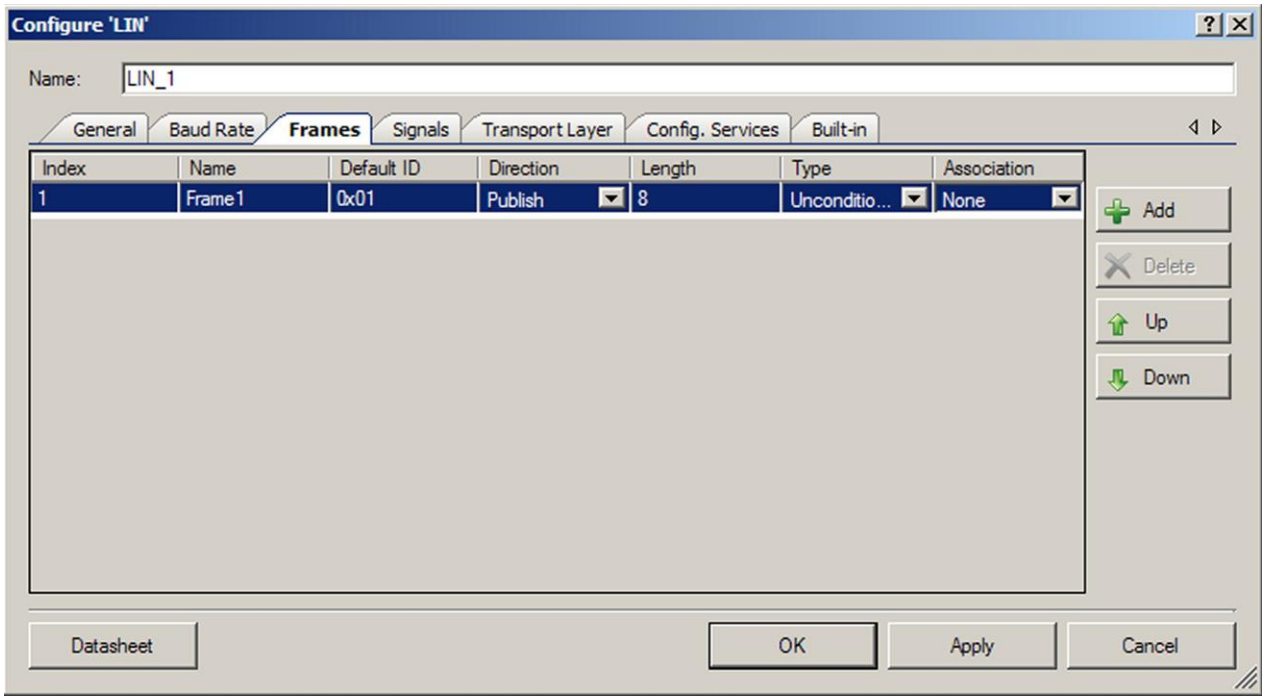
The actual value of the bus baud rate is displayed here. The LIN slave will work on this baud rate. The BusClk value can be modified to make **Nominal LIN Bus Baud Rate** equal to **Actual LIN Bus Baud Rate**.

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.

Figure 11. Configure LIN Dialog, Frames Tab



Frame Configuration Table

The configuration table is situated in the middle of this tab. The 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.

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.

The **Message ID** column is not shown in Figure 11. This is because it 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.

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.

Tab Buttons

There are four buttons available on this tab.

The **Add** button adds a new frame to the table.

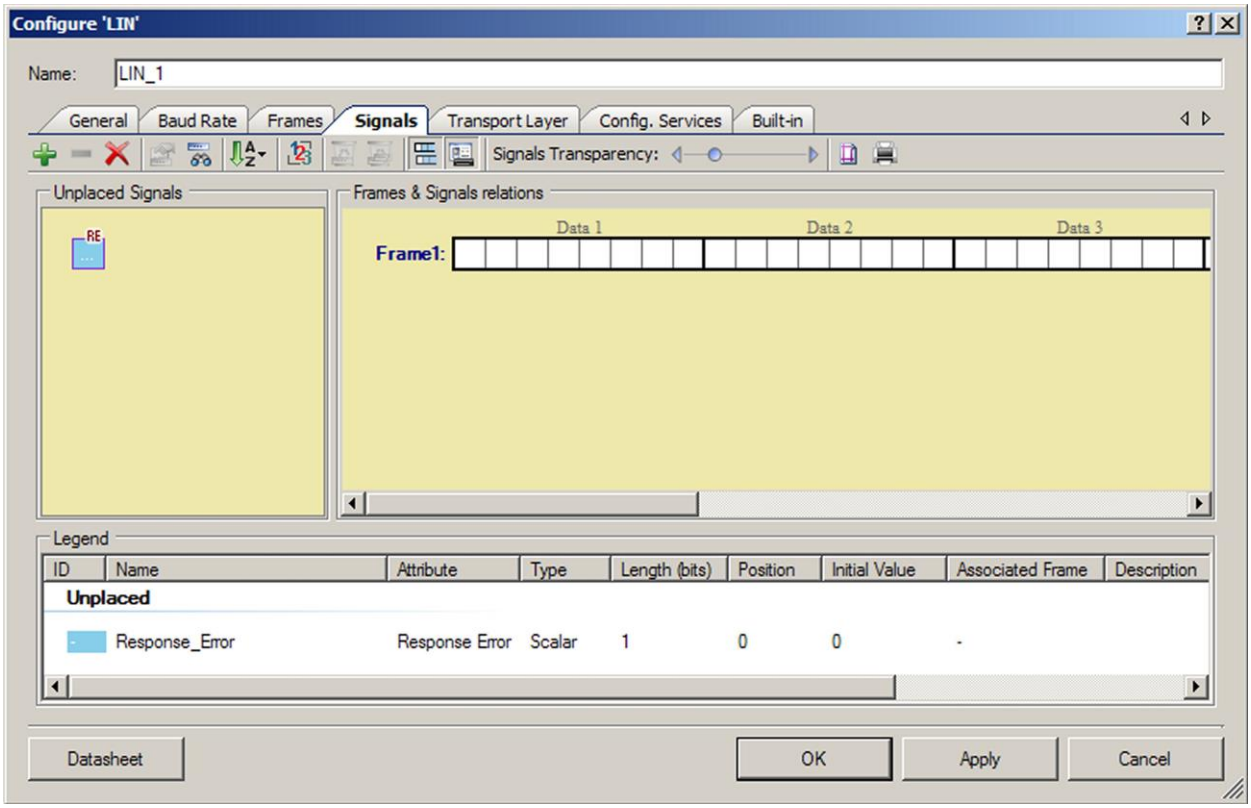
The **Delete** button deletes the currently selected frame from the table. The index number fields are changed accordingly. If a frame is deleted on this tab, any signals that are packed into it (configured with the **Signals** tab) are moved into the **Unplaced Signals** region (See Sort Signals button in the Signals Tab section).

You can use the **Up** and **Down** buttons to reorder the **Index** number values for each frame.

Signals Tab

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

Figure 12. Configure LIN Dialog, Signals Tab



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 (see Figure 12). 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 2.7.3 “Reporting to the Cluster” of the LIN 2.1 specification.

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.

Figure 13. Signals Toolbar



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. **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.

3. **Find Signal** button

   This button allows you to search for a certain signal.

4. **Sort Signals** button

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

5. **Renumber Signals** button

   This button renumbers the signal index values in ascending order.

6. **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.

7. **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.

8. **Show/Hide Legend** button

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

9. **Signals Transparency** slider

   This slider sets the transparency for signals graphics.

10. **Print** buttons

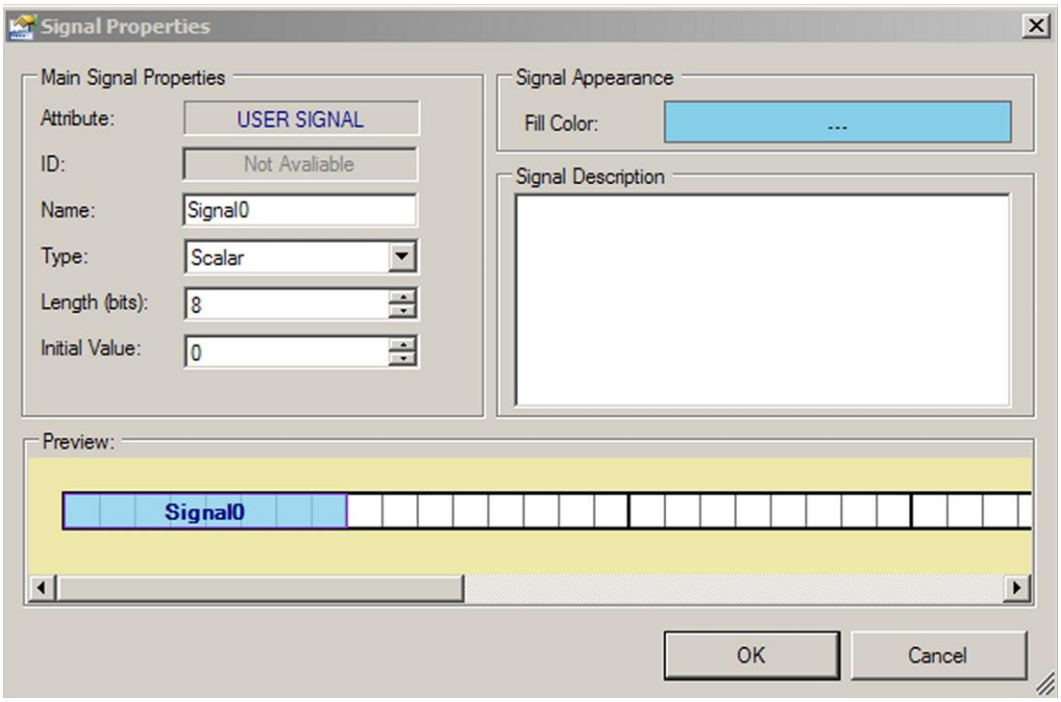
    These buttons print out the **Frames & Signals relations** region.

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 14). 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 14. Signal Properties Window



Name

The **Name** property is used to choose the name of the signal. The default signal name is Signalx, 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.

Type

This property is used to select the type of the signal. There are two types of signal, as defined in the LIN 2.1 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.

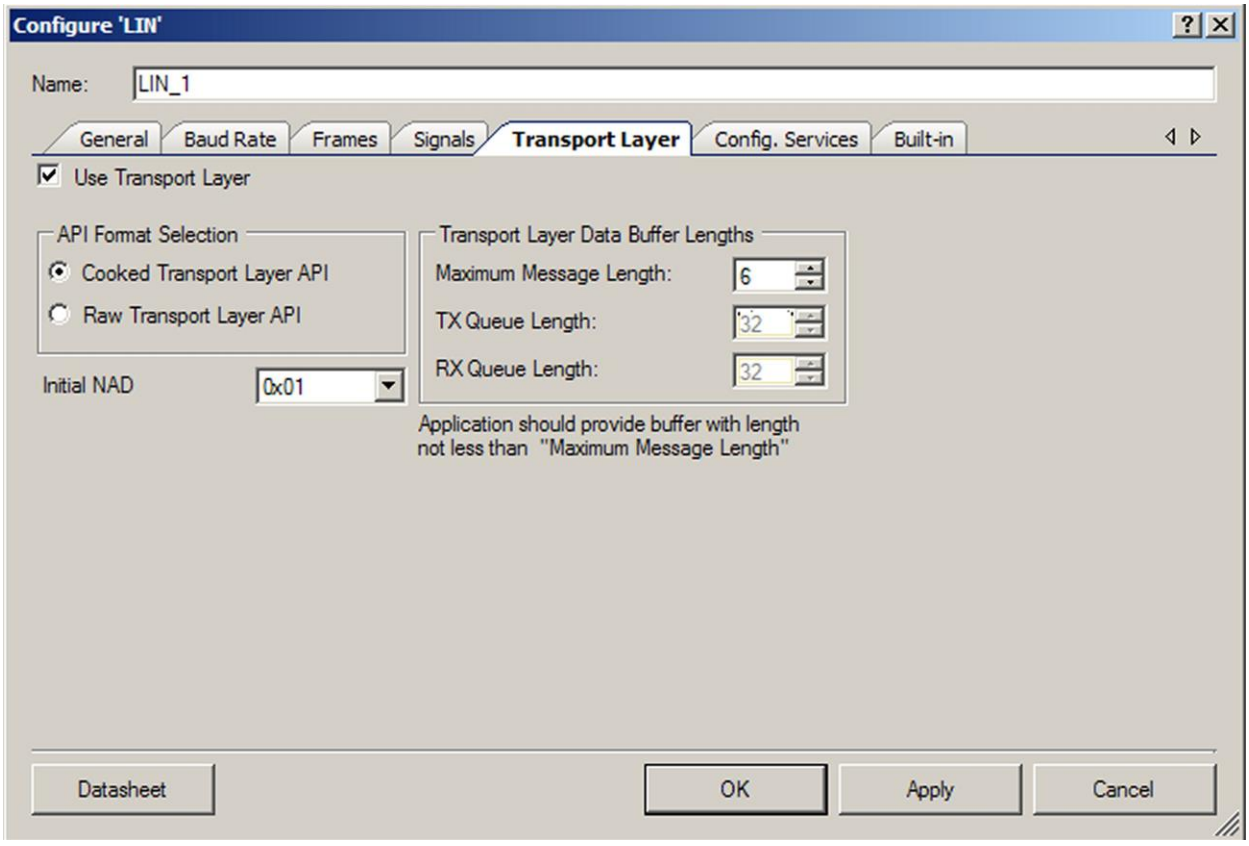
Preview

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

Transport Layer Tab

The Transport Layer tab view is shown in Figure 15.

Figure 15. Configure LIN Dialog, 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 LIN 2.1 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. 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 the LIN 2.1 specification in section 7.4.

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 J2602-1 Compliance checkbox 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	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

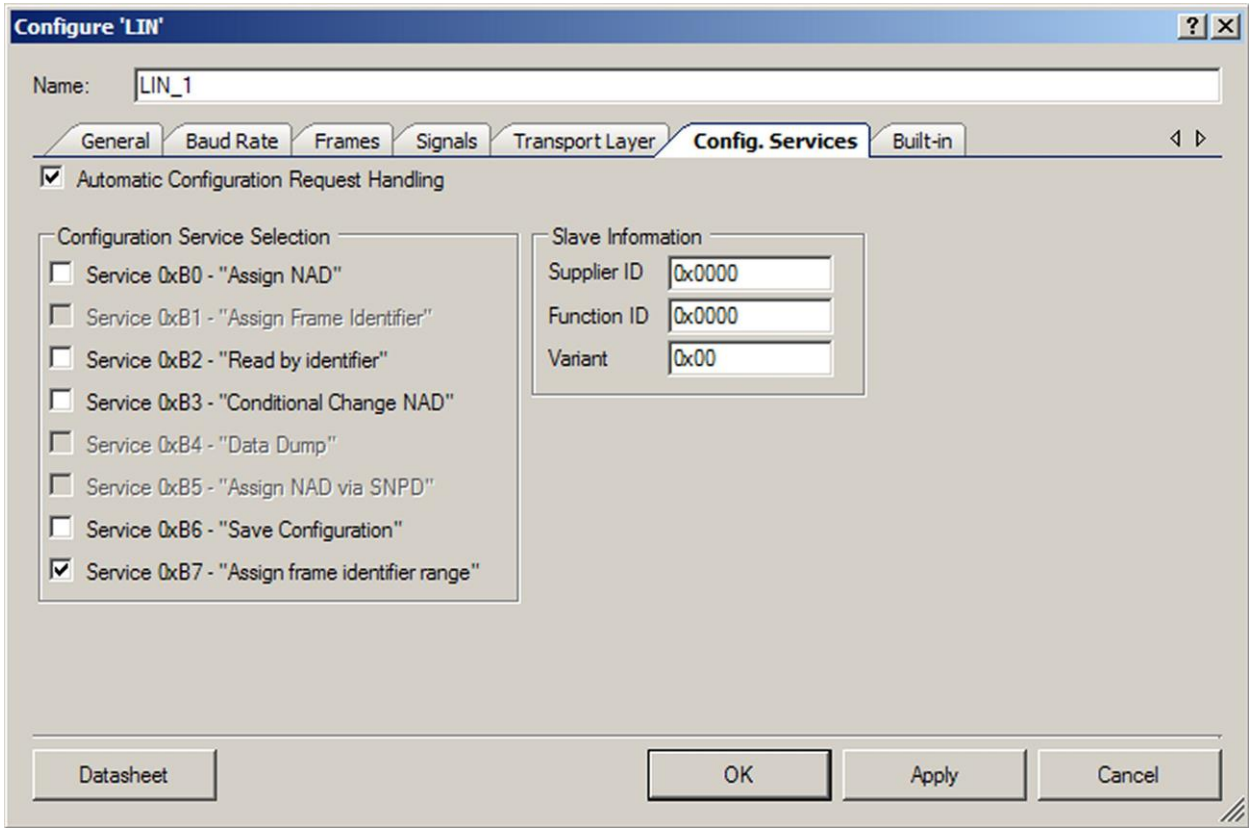
These properties are only applicable when the **Raw Transport Layer API** format is selected. When using the raw API format, there is a message “queue” that buffers the frame response data that is being sent or received. If the slave cannot update the queues very quickly, then the queue lengths should be made longer. If the slave can update the queues very quickly, then the queues can be made shorter to decrease RAM 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 LIN 2.1 specification defines Configuration Service requests that the slave must support (some are mandatory and some are optional with regard to the LIN 2.1 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 4.6 of the LIN 2.1 specification. This component supports some of them. You have the option of disabling or enabling each of the supported services individually. The configuration service requests are described in section 4.2.5 of the LIN 2.1 specification.

Figure 16. Configure LIN Dialog, Configuration Services Tab



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 selected, all of the other options on the tab are available. If the box is not selected, 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 LIN 2.1 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 LIN 2.1 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 not described in the LIN 2.1 specification. It is only 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 LIN 2.1 specification. This request is used to allow the LIN master to read the slave's identification information (Supplier ID, Function ID, Variant). This component only supports the LIN Product Identification version of this request.

- Service 0xB3 – “Conditional Change NAD”

  This is an optional service in the LIN 2.1 specification.

  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 request is optional in the LIN 2.1 specification and is not supported by this component.

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

  “Assign NAD via SNPD” (0xB5) service is not supported by the LIN 2.1 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 LIN 2.1 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 LIN 2.1 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 0xFFFE. 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 ±1.5 percent when the **Automatic Baud Rate Synchronization** option is disabled and ±14 percent when it is not. A warning will be displayed if the clock cannot be generated within this limit. In this case, you should modify the Master Clock source in the DWR.

Placement

Only one component instance can be placed per design.

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
l_sys_init()	Initializes the LIN core.

Signal Interaction Functions Subgroup

Function	Description
l_bool_rd()	Reads and returns the current value of the signal for one-bit signals.
l_u8_rd()	Reads and returns the current value of the signal for signals of two to eight bits.
l_u16_rd()	Reads and returns the current value of the signal for signals of 9 to 16 bits.
l_bytes_rd()	Reads and returns the current values of the selected bytes in the signal.
l_bool_w()	Sets the current value of the signal for one-bit signals to v.
l_u8_wr()	Sets the current value of the signal for signals of two to eight bits.
l_u16_wr()	Sets the current value of the signal for signals of 9 to 16 bits.
l_bytes_wr()	Sets the current values of the selected bytes in the signal.

Notification Functions Subgroup

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

Interface Management Functions* Subgroup

Function	Description
l_ifc_init()	Initializes the LIN Slave component.
l_ifc_wake_up()	Transmits one wakeup signal.
l_ifc_ioctl()	Controls functionality beyond the specification.
l_ifc_rx()	The LIN Slave component calls this API routine automatically.
l_ifc_tx()	The LIN Slave component calls this API routine automatically.
l_ifc_aux()	The LIN Slave component calls this API routine automatically.
l_ifc_read_status()	Returns the status of the specified LIN interface.

User-Provided Callouts Subgroup

Function	Description
l_sys_irq_disable()	Disables all interrupts for the component.
l_sys_irq_restore()	Restores all interrupts for the component.

Node Configuration Functions

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

Transport Layer Functions

Initialization Subgroup

Function	Description
ld_init()	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.

Raw Transport Layer API Functions Subgroup

Function	Description
ld_put_raw()	The call queues the transmission of 8 bytes of data in one frame.

Function	Description
ld_get_raw()	Copies the oldest received diagnostic frame data to the memory specified by input parameter.
ld_raw_tx_status()	Returns the status of the raw frame transmission function.
ld_raw_rx_status()	Returns the status of the raw frame receive function

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().

Non-LIN-Specified API

Function	Description
LIN_Start()	Starts the component operation.
LIN_Stop()	Stops the component operation.

Core API Functions: Initialization

l_bool l_sys_init()

Description: Initializes the LIN core. This function does nothing and always returns zero.

Static Prototype: l_bool l_sys_init(void)

Dynamic Prototype: None

Parameters: None

Return Value: Always returns zero.

Side Effects: None

Core API Functions: Signal Interaction

In all static signal API calls that follow, the “sss” is the name of the signal, for example, l_u8_rd_EngineSpeed(). For dynamic signal API calls that follow, the “sss” is a signal handle, as defined in Application Programming Interface.

l_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.
Static Prototype:	l_bool l_bool_rd_sss(void)
Dynamic Prototype:	l_bool l_bool_rd(l_signal_handle sss)
Parameters:	sss: Signal handle of the signal to read.
Return Value:	Returns the current value of the signal.
Side Effects:	None

l_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.
Static Prototype:	l_u8 l_u8_rd_sss(void)
Dynamic Prototype:	l_u8 l_u8_rd(l_signal_handle sss)
Parameters:	sss: Signal handle of the signal to read
Return Value:	Returns the current value of the signal.
Side Effects:	None

l_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.
Static Prototype:	l_u16 l_u16_rd_sss(void)
Dynamic Prototype:	l_u16 l_u16_rd(l_signal_handle sss)
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.

l_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:	void l_bytes_rd_sss(l_u8 start, l_u8 count, l_u8* const data)
Dynamic Prototype:	void l_bytes_rd(l_signal_handle sss, l_u8 start, l_u8 count, l_u8* const data)
Parameters:	sss: Signal handle of the signal to read start: First byte to read from count: Number of bytes to read data: Pointer to array, in which the data read from the signal is stored
Return Value:	None
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.

l_bool_wr()

Description:	Writes the value v to the signal. If an invalid signal handle is passed into the function, no action is taken.
Static Prototype:	void l_bool_wr_sss(l_bool v)
Dynamic Prototype:	void l_bool_wr(l_signal_handle sss, l_bool v)
Parameters:	sss: Signal handle of the signal to write v: Value of the signal to be set
Return Value:	None
Side Effects:	None

l_u8_wr()

Description: Writes the value v to the signal. If an invalid signal handle is passed into the function, no action is taken.

Static Prototype: void l_u8_wr_sss(l_u8 v)

Dynamic Prototype: void l_u8_wr(l_signal_handle sss, l_u8 v)

Parameters: sss: Signal handle of the signal to write

v: Value of the signal to be set

Return Value: None

Side Effects: None

l_u16_wr()

Description: Writes the value v to the signal. If an invalid signal handle is passed into the function, no action is taken.

Static Prototype: void l_u16_wr_sss(l_u16 v)

Dynamic Prototype: void l_u16_wr(l_signal_handle sss, l_u16 v)

Parameters: sss: Signal handle of the signal to write;

v: Value of the signal to be set.

Return Value: None

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.

l_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 l_bytes_wr_sss(l_u8 start, l_u8 count, const l_u8* const data)
Dynamic Prototype:	void l_bytes_wr(l_signal_handle sss, l_u8 start, l_u8 count, const l_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
Return Value:	None
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.

Core API Functions: Notification

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, l_flg_tst_RxEngineSpeed(). For the dynamic flag API routines the “fff” is a signal handle, as defined earlier in Application Programming Interface.

l_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:	l_bool l_flg_tst_fff(void)
Dynamic Prototype:	l_bool l_flg_tst(l_flag_handle fff)
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.
Side Effects:	None

l_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 l_flg_tst() API). The component does not automatically clear notification flags. This routine is the only way that a notification flag can be cleared.
Static Prototype:	void l_flg_clr_fff(void)
Dynamic Prototype:	void l_flg_clr(l_flag_handle fff)
Parameters:	fff: Name of the flag handle
Return Value:	None
Side Effects:	None

Interface Management Functions

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 API call, for example, l_ifc_init_MyLinIfc(). For this component, the interface name is the same as the component instance name. This component supports a maximum of one interface. Therefore, there will never be more than one valid identifier for “iii.”

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.

Side Effects: None

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

Return Value: None

Side Effects: None

l_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_ioctl_op op, void* pv)

Dynamic Prototype: l_u16 l_ifc_ioctl(l_ifc_handle iii, l_ioctl_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
L_IOCTL_READ_STATUS	0x00u	None	Optional status indicators
L_IOCTL_SET_BAUD_RATE	0x01u	l_u16 l_u16	Modify baud rate
L_IOCTL_SLEEP	0x02u	None	Prepare device for low-power-mode entry
L_IOCTL_WAKEUP	0x03u	None	Restore component state after wakeup
L_IOCTL_SYNC_COUNTS	0x04u	None	Return current number of sync field timer counts
L_IOCTL_SET_SERIAL_NUMBER	0x05u	l_u8*	Update the pointer to the serial number

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 J2602-1 Compliance is enabled).

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 it 8 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.

Side Effects: None

l_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 l_ifc_rx_iii(void)

Dynamic Prototype: void l_ifc_rx(l_ifc_handle iii)

Parameters: iii: Name of the interface handle

Return Value: None

Side Effects: None

l_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 l_ifc_tx_iii(void)

Dynamic Prototype: void l_ifc_tx(l_ifc_handle iii)

Parameters: iii: Name of the interface handle

Return Value: None

Side Effects: None

l_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 l_ifc_aux_iii(void)

Dynamic Prototype: void l_ifc_aux(l_ifc_handle iii)

Parameters: iii: Name of the interface handle

Return Value: None

Side Effects: None

l_ifc_read_status()

Description: This function returns the status of the previous communication. Refer to the LIN 2.1 specification for detailed information on each status information field in the LIN Slave status word.

Static Prototype: l_u16 l_ifc_read_status_iii(void)

Dynamic Prototype: l_u16 l_ifc_read_status(l_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 configu ration	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.

Side Effects: None

User-Provided Callouts

l_sys_irq_disable()

Description: This function disables all interrupts for the component. It returns a mask of the state that the interruptmask bits were in. This function is essentially equivalent to the DisableInt API of most components.

Unlike those functions, the returned value must be saved and later used with the l_sys_irq_restore() function to restore the interrupt state properly. It is highly recommended that you be very careful when using this API routine. It is likely that LIN communication failures will occur if the interrupts for this component are disabled for too long.

This routine is supposed to be provided by the application. However, the LIN Slave component implements this routine automatically. You can modify the code in the routine if necessary.

Static Prototype: None

Dynamic Prototype: l_irqmask l_sys_irq_disable(void)

Parameters:

Return Value: Returns an interrupt register mask that defines the digital blocks for which interrupts were disabled.

Side Effects: None

l_sys_irq_restore()

Description: This function restores interrupts for the component. It should be used in conjunction with l_sys_irq_disable(). This function is essentially equivalent to the EnableInt API of most components; however, it should not be called when the component is being started.

This routine is supposed to be provided by the application. However, the LIN Slave component implements this routine automatically. You can modify the code in the routine if necessary.

Static Prototype: None

Dynamic Prototype: void l_sys_irq_restore(l_irqmask previous)

Parameters: previous: Interrupt mask that defines the digital blocks for which interrupts will be enabled.

Return Value: None

Side Effects: None

Node Configuration Functions

ld_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.

Static Prototype: None

Dynamic Prototype: l_u8 ld_read_configuration(l_ifc_handle iii, l_u8* const data, l_u8* length)

Parameters: iii: Name of the interface handle;

data: 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

Side Effects: None

ld_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 ld_read_configuration function() for information on what the configuration data contains and how it is stored.

Static Prototype: None

Dynamic Prototype: l_u8 ld_set_configuration(l_ifc_handle iii, const l_u8* const data, l_u16 length)

Parameters: iii: Name of the interface handle

data: 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

Side Effects: None

ld_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.

Static Prototype: None

Dynamic Prototype: l_u8 ld_read_by_id_callout (l_ifc_handle iii, l_u8 id, l_u8* data)

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 set manually. If set, it specifies that no response will be provided for the service.
LD_POSITIVE_RESPONSE	You can set this status set manually. If set, it specifies that response will be provided for the service. The response will be pointed by the frameData parameter.

Side Effects: None

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.

ld_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 be aborted; instead, the API waits until the message is completed.

Static Prototype: None

Dynamic Prototype: void ld_init(l_ifc_handle iii)

Parameters: iii: Name of the interface handle

Return Value: None

Side Effects: None

Raw Transport Layer API Functions

ld_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.

Static Prototype: None

Dynamic Prototype: void ld_put_raw(l_ifc_handle iii, const l_u8* const data)

Parameters: iii: Name of the interface handle

data: Array of data to be sent

Return Value: None

Side Effects: None

ld_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.

Static Prototype: None

Dynamic Prototype: void ld_get_raw(l_ifc_handle iii, l_u8* const data)

Parameters: iii: Name of the interface handle

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

Return Value: None

Side Effects: None

ld_raw_tx_status()

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

Static Prototype: None

Dynamic Prototype: l_u8 ld_raw_tx_status(l_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 ld_put_raw() 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.

Side Effects: None

ld_raw_rx_status()

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

Static Prototype: None

Dynamic Prototype: l_u8 ld_raw_rx_status(l_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.

Side Effects: None

Cooked Transport Layer API Functions

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.

Static Prototype: None

Dynamic Prototype: void ld_send_message(l_ifc_handle iii, l_u16 length, l_u8 nad, const l_u8* const 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

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

Return Value: None

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.

ld_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.

Static Prototype: None

Dynamic Prototype: void ld_receive_message(l_ifc_handle iii, l_u16* const length, l_u8* const nad, l_u8* const 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

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

Return Value: None

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 ld_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.

Static Prototype: None

Dynamic Prototype: l_u8 ld_tx_status(l_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 ld_send_message 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 l_read_status().
LD_N_AS_TIMEOUT	The transmission failed because of an N_As timeout, and current message transmission will be aborted. See Section 3.2.5 of the LIN 2.1 specification.

Side Effects: None

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.

Static Prototype: None

Dynamic Prototype: l_u8 ld_rx_status(l_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 reception is not yet completed.
LD_COMPLETED	The reception has completed successfully and all information (length, NAD, data) is available. You can also issue a new ld_receive_message() call. This value is also returned after initialization of the transport layer.
LD_FAILED	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 l_read_status().
LD_N_CR_TIMEOUT	The reception failed because of an N_Cr timeout, and current message reception will be aborted. See Section 3.2.5 of the LIN 2.1 specification.
LD_WRONG_SN	The reception failed because of an unexpected sequence number.

Side Effects: None

Non-LIN-Specified API

LIN_Start()

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

Static Prototype: None

Dynamic Prototype: l_bool LIN_Start()

Parameters: None

Return Value: Zero: The initialization succeeded.
Nonzero: The initialization failed.

Side Effects: None

LIN_Stop()

Description:	Stops the component operation. This function is not required.
Static Prototype:	None
Dynamic Prototype:	l_bool LIN_Stop()
Parameters:	None
Return Value:	None
Side Effects:	None

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.

PSoC 3 Reentrancy Support

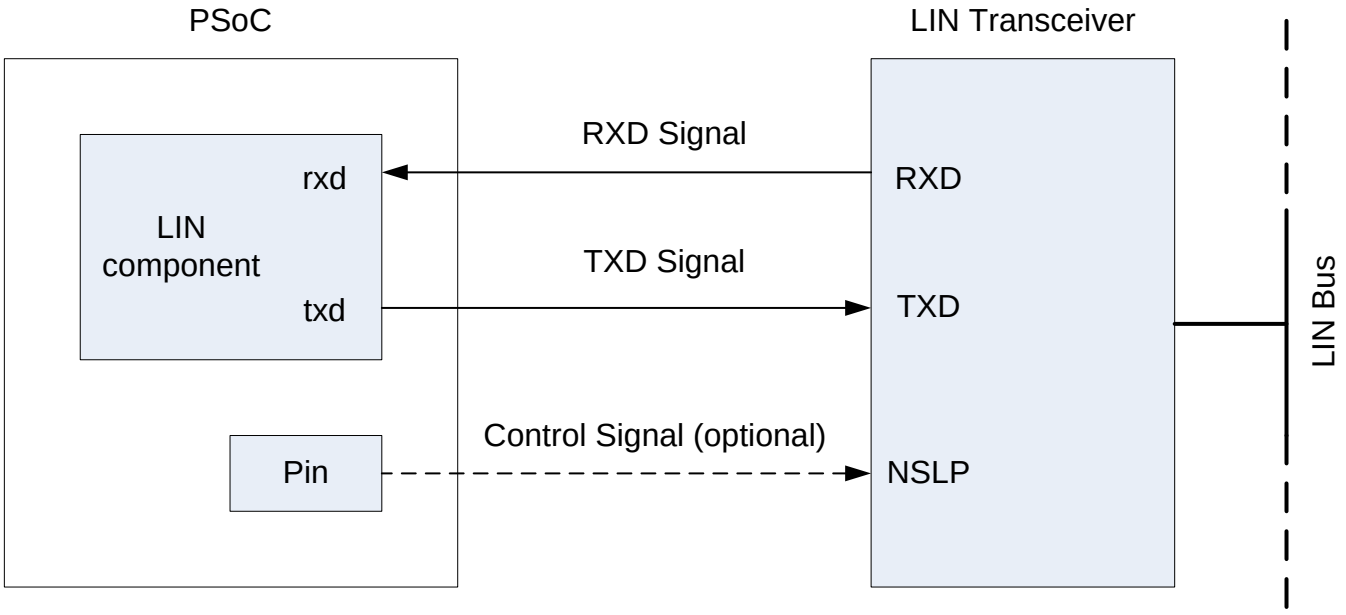
The CyIntClearPending() function can be concurrently called because it is called from two different interrupts inside of the LIN component. While not reentrant by default, it can be made to support reentrancy to eliminate “MULTIPLE CALL TO FUNCTION” warning during compilation. Refer to the “Reentrant Code in PSoC 3” topic in the PSoC Creator Help for more information. Also, the component example project has reentrancy support added.

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.

Figure 17. Hardware Interface between PSoC and LIN Bus



Resources

The LIN component is placed throughout the UDB array. The component utilizes the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
LIN_Slave_Example project	4	42	3	3	–	2

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 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
LIN_Slave_Example project	5694	153	4706	198	4530	190

DC and AC Electrical Characteristics

For information about DC and AC Electrical Characteristics refer to the “LIN Physical Layer Specification” chapter of the *LIN 2.1 Specification*.

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted. Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics

Parameter	Description	Min	Typ[1]	Max	Units
I_{DD}	Component current consumption	–	190	–	µA

1. Device IO and clock distribution current not included. The values are at 25 °C.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
1.10.a	Minor datasheet edit.	
1.10	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. 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, 2012-2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.