LIN Bus Slave Datasheet LINS V 3.00

Copyright © 2010-2013 Cypress Semiconductor Corporation. All Rights Reserved.

Resource Usage

Setting	Parameter Value	API Memory (Bytes)		PSoC® Blocks			Pins
		Flash	RAM	Digital	Analog CT	Analog SC	
CY8C21345-12PVXE, CY8C21345-24PVXA, CY8C21645-12PVXE, CY8C21645-24PVXA, CY8C22345-12PVXE, CY8C22345-24PVXA, CY8C22345H-24PVXA, CY8C22645-12PVXE, CY8C22645-24PVXA, CY8C24x23A-24xxXA, CY8C24x23A-12xxXE, CY8C24x94-24xxXA, CY8C21x34-24xxXA, CY8C21x34-12xxXE, CY8C29466-12PVXE, CY8C29466-24PVXA, CY8C29666-24PVXA, CY8C29666-12PVXE							
LIN Slave Core	Auto Baud Rate Sync Enabled	1624	31	3	0	0	2
	Auto Baud Rate Sync Disabled	1498	27				
Flags*	Unique and duplicated signals count, total frames count	Equation 1 For Dynamic API: 77	Equation 2				
Frames	Unconditional and total frames count, length of each frame	Equation 3	Equation 4				
Core API	Static	SignalsCount X (30..90)**	SignalsCount X (0..2)**				
	Dynamic	SignalsCount X (25..90)**	SignalsCount X (0..2)**				
	Both	Static + Dynamic	SignalsCount X (0..2) + (0..2)				
Transport Layer	Cooked API	1070	41				
	Raw API	719	28 + RxQueue Length + TxQueue Length				

Resource Usage (continued)

Setting	Parameter Value	API Memory (Bytes)		PSoC® Blocks			Pins
		Flash	RAM	Digital	Analog CT	Analog SC	
CY8C21345-12PVXE, CY8C21345-24PVXA, CY8C21645-12PVXE, CY8C21645-24PVXA, CY8C22345-12PVXE, CY8C22345-24PVXA, CY8C22345H-24PVXA, CY8C22645-12PVXE, CY8C22645-24PVXA							
Configuration Services	Services: 0xB0, 0xB2, 0xB3, 0xB6, 0xB7	452	0				
	Service: 0xB1 (LIN2.0 Compatibility)	125	0				
LIN2.0 Compatibility	Enabled	2	0				
J602 Compliance	Enabled	423	1				

Notes

* There are different flags for each data unit. Each flag has one RAM bit, and in the case of static API, has also two access functions. In case of dynamic API, flag accessing functions are common for all flags.

** Depending on signal type and position.

$$(\text{UniqueSignalsCount} + \text{DuplicatedSignalsCount} + \text{FramesCount}) \times 20$$

Equation 1

$$\frac{\text{UniqueSignalsCount} + \text{DuplicatedSignalsCount} + \text{FramesCount}}{8}$$

Equation 2

$$F_{cnt} + F_{cnt} \times 2 \times \text{LIN2.0_COMPL} + F_{cnt} \times 3 + \sum_{i=0}^{UF_{cnt}-1} UF_{length}[i]$$

Equation 3

$$\sum_{i=0}^{UF_{cnt}-1} UF_{length}[i] + F_{cnt}$$

Equation 4

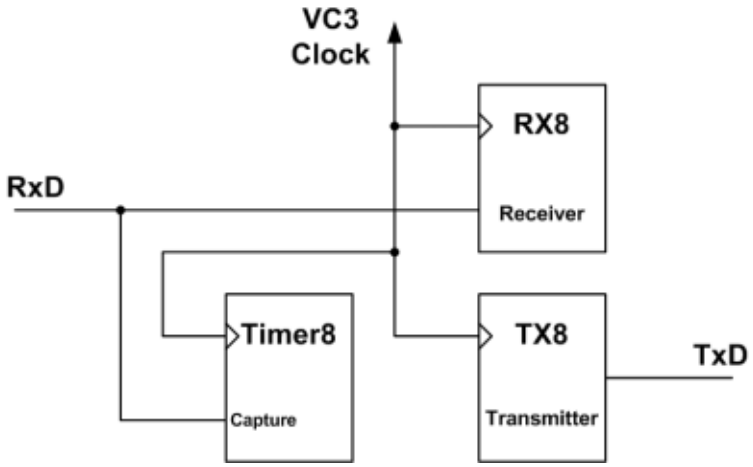
Features and Overview

- 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
- Wizard for fast and easy configuration
- Import of *.ncf/*.ldf files and *.ncf file export
- Editor for *.ncf/*.ldf files with syntax checking

The LINS User Module implements a LIN 2.1 slave node on a PSoC device. In addition, options for LIN 2.0 or SAE J2602-1 compliance are available. This user module 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 user module provides an API that conforms to the API specified by the LIN 2.1 specification.

This user module provides a good combination of flexibility and ease-of-use. A wizard for the user module is provided that allows configuring all parameters of the LIN slave in an easy manner.

Figure 1. LINS Block Diagram



Functional Description

The LINS User Module uses digital PSoC blocks combined with code to implement a LIN 2.1 slave node in a PSoC device.

The digital blocks generate user module interrupts that take care of all of the tasks listed in the previous section. A LIN 2.1 compliant API is built on top of the hardware and ISRs of the user module.

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.

Automatic Baud Synchronization

Automatic Baud Synchronization refers to the feature of the LIN slave that uses the Sync Byte Field to measure the baud rate of the bus, and subsequently adjusts a clock divider to lock on to the baud rate. If this feature is disabled, the slave does not measure the baud rate from the Sync Byte Field. If Automatic Baud Rate synchronization is disabled, the slave must have a more accurate clock than the normal IMO accuracy. If Automatic Baud Rate synchronization is enabled, the normal PSoC1 IMO accuracy is acceptable.

Break Field

Refer section 2.3.1.1 of the LIN 2.1 specification.

Continuing Frame (CF)

Refer section 3.2.1 of the LIN 2.1 specification.

Cooked API

Refer section 7.4.1 of the LIN 2.1 specification.

Dynamic API

Refer section 7.1.1.1 of the LIN 2.1 specification.

First Frame (FF)

Refer section 3.2.1 of the LIN 2.1 specification.

Frame

Refer section 2.3 of the LIN 2.1 specification.

Frame PID

Refer section 2.3.1.3 of the LIN 2.1 specification.

Frame ID

Refer section 2.3.1.3 of the LIN 2.1 specification.

Master Request Frame (MRF)

This is the frame with unique ID. It always transports data from the master to slaves. It is used for Diagnostic requests, Configuration service requests, and Transport Layer messages.

Raw API

Refer section 7.4.1 of the LIN 2.1 specification.

Signal

Refer section 2.2 of the LIN 2.1 specification.

Single Frame (SF)

Refer section 3.2.1 of the LIN 2.1 specification.

Slave Response Frame (SRF)

This is the frame with unique ID. It always transports data from a slave to the master. It is used for Diagnostic responses, Configuration service responses, and Transport Layer response messages.

Static API

Refer section 7.1.1.1 of the LIN 2.1 specification.

Sync Byte Field

Refer section 2.3.1.2 of the LIN 2.1 specification.

Transport Layer

Refer section 7.4 of the LIN 2.1 specification.

Block Resources

The LINS User Module maps onto three PSoC blocks designated RX8, TX8, and Timer8, in the PSoC Designer Device Editor. The individual blocks are:

RX8 Block

This block has multiple purposes for the LINS User Module. This block is primarily used to receive the Protected ID (PID) of all frame headers and any frame responses that this slave node subscribes to.

This block is also used to receive any data that the user module transmits with the TX8 block. After the transmitted data is also received (“read back”), the data received is compared with the transmitted data to check if any error has occurred. The RX8 block also checks for a framing error in the bytes that it “reads back”.

This block is always placed in a DCx3 digital block.

TX8 Block

This block has multiple purposes for the LINS User Module. This block is primarily used to transmit data when the slave node is the publisher of a frame response.

The second purpose of this block is to serve as a timer. This block is reconfigured as a timer whenever the slave is not actively transmitting data. So, this block only functions as a transmitter when the slave has received a PID that specifies that the slave should send a frame response. At this point, the block is converted into a transmitter to transmit the response. When this block functions as a timer, its purpose is to check to see if the time of bus inactivity (no signal transitions) is too long. In this case, a "bus inactivity" flag is raised to the application.

The third purpose of this block is to serve as a wake up source for the device when the user module (along with the device) has been put into a sleep state. In this case, the block is reconfigured to generate an interrupt when a falling edge at the block's input occurs. When the falling edge occurs, the block generates a hardware interrupt signal that wakes the device up. This "falling edge interrupt" is generated whenever a wake up signal occurs on the LIN bus.

This block is always placed in a DCx2 digital block.

Timer8 Block

This block is used for multiple purposes for the LINS User Module. This block is primarily used to measure the time of all dominant (that is, LOW) pulses on the LIN bus at all times. The Timer8 block generates a "capture" interrupt on every edge (rising and falling) of the LIN bus. The time between each falling and rising edge is calculated. If the measured time of the dominant pulse exceeds a certain threshold, then it is detected as a break field on the LIN bus.

The second purpose of the Timer8 block is to measure the baud rate of the bus from the sync byte field of a LIN frame header. This measured value is used to adjust the VC3 divider value. This is for the purpose of automatic baud rate synchronization.

This block can be placed in either a DBx0 or DBx1 digital block of the digital row that the user module occupies. You can modify the placement of this block to be either one of the two specified.

VCx Clock(s)

This user module is always clocked by VC3. When automatic baud rate synchronization is used, this user module may change the VC3 source and divider value at run-time. As a result, VC3 should not be used to clock anything else other than the LINS User Module when automatic baud rate synchronization is used. In addition, application code must not modify the VC3 clock's source or divider value.

If automatic baud rate synchronization is not used, then other resources in the device can use VC3 as a clock source. However, the application code must not modify VC3's source or divider value.

VC3 is used to clock the RX8, TX8, and Timer8 blocks. VC3 normally has a frequency that provides the nominal baud rate on the LIN bus. However, during sync byte field measurement, the VC3 clock divider must be modified to increase the frequency of VC3.

This user module may also use VC1, depending on the baud rate settings for the user module. However, the user module does not modify the VC1 divider value at run-time, so you can use VC1 to clock other resources. However, you must not modify the VC1 divider value with application code.

Important Design Considerations

Software Interrupts Compatibility

WARNING: The LINS User Module overwrites the contents of its associated INT_CLRx register with logical '1' values during its operation. This means that the Software Interrupts feature cannot be used along with the LINS User Module. As a result, do not alter the value of the ENSWINT bitfield in the INT_MSK3 register - this will break project operation.

Interrupt Latency

The period between two consecutive Timer8 block interrupts during the frame response should not exceed the value given by Equation 5. If the value given by Equation 5 is not met due to long interrupt duration of other sources, the frame transaction can be incomplete.

Time period between Timer8 block interrupts is calculated as shown in Equation 5:

$$\text{Time between Timer interrupts} = \frac{1}{8 \times \text{BaudRate}}$$

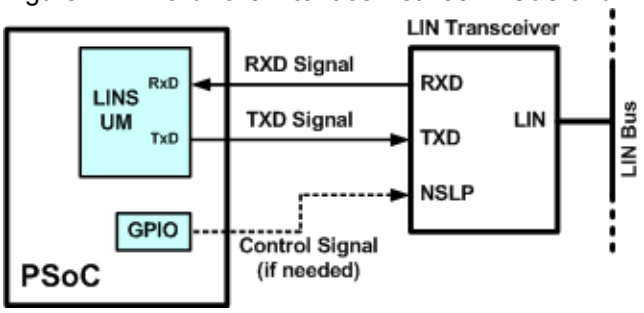
Equation 5

PSoC and LIN Bus Hardware Interface

A LIN physical layer transceiver device is needed when the PSoC LIN slave node is connected directly to a LIN bus. In this case, the TxD pin of the LINS User Module 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 needed, 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. This user module does not provide this control signal on the PSoC automatically. Instead, a CPU-controlled GPIO pin should be used to output the desired signal to the LIN transceiver device if this signal is needed.

Figure 2. Hardware Interface Between PSoC and LIN Bus



DC and AC Electrical Characteristics

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

Placement

The LINS User Module occupies three digital PSoC blocks and is restricted to only one instance in a project.

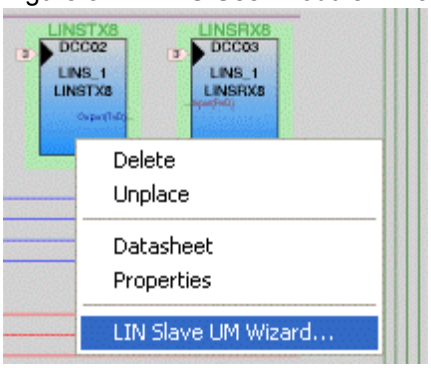
The wizard for this user module configures digital routing to properly connect the user module to the pins that you have selected in the wizard.

The LINS User Module comprises of TX8, RX8, and Timer8 blocks. All these blocks must be placed on the same digital row. However, you can place this user module on any digital row in the device.

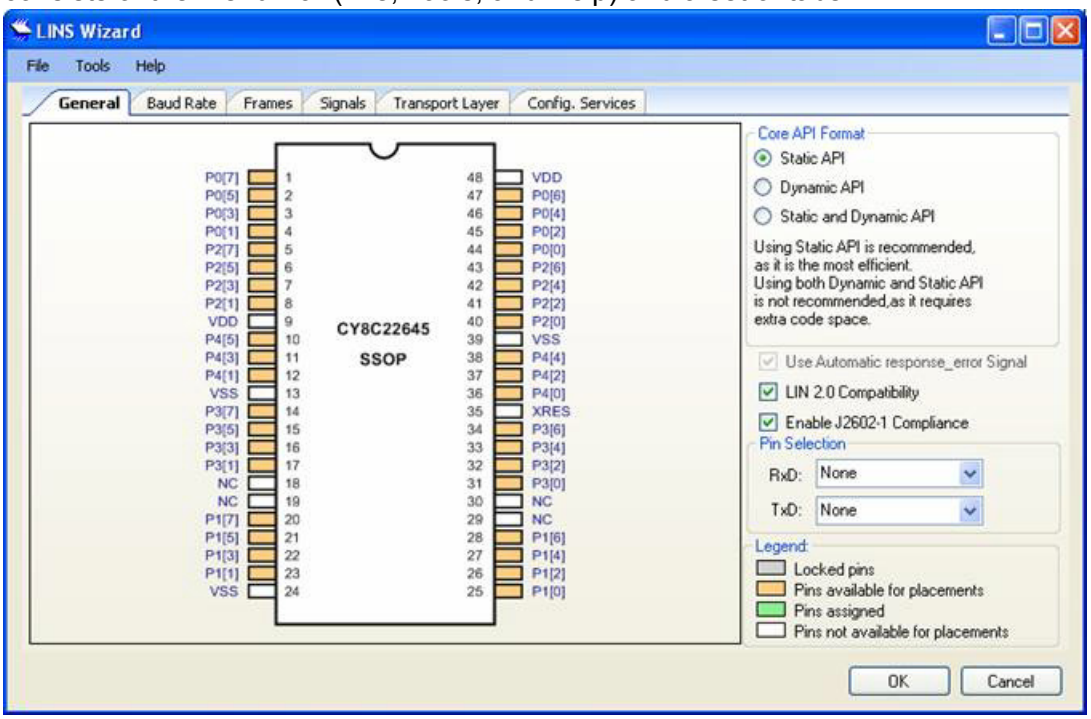
Wizard

To access the Wizard, right click the user module in the Workspace Explorer or the Chip Editor and select the LINS Wizard (Figure 3).

Figure 3. LINS User Module Wizard Access



The LINS User Module Wizard consists of multiple tabs that each provide configuration settings for a particular aspect of the LINS User Module. The following screenshot shows the Wizard Main Form that consists of the Menu Bar (File, Tools, and Help) and a set of tabs.



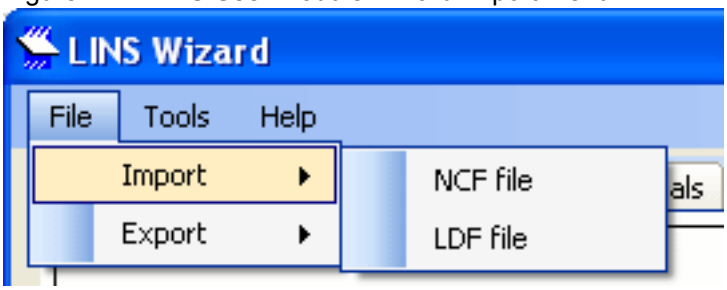
LINS Wizard Menu

File Menu

Import

File > Import menu enables you to import a LIN Description File (LDF) or a Node Capability File (NCF) (see Figure 4).

Figure 4. LINS User Module Wizard Import Menu

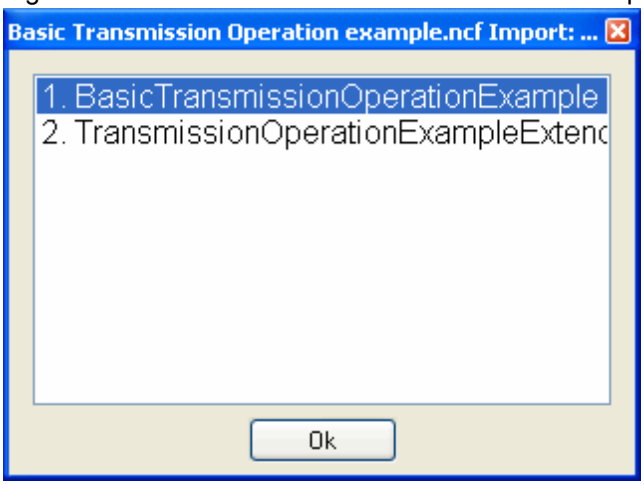


An imported file configures the wizard and user module settings to match the configuration of the node that was selected from the list of the existing nodes of the NCF/LDF file.

If the syntax in imported file was correct, a list of available nodes is displayed. A similar list is shown in Figure 5. Choose one of the available node descriptions to import from.

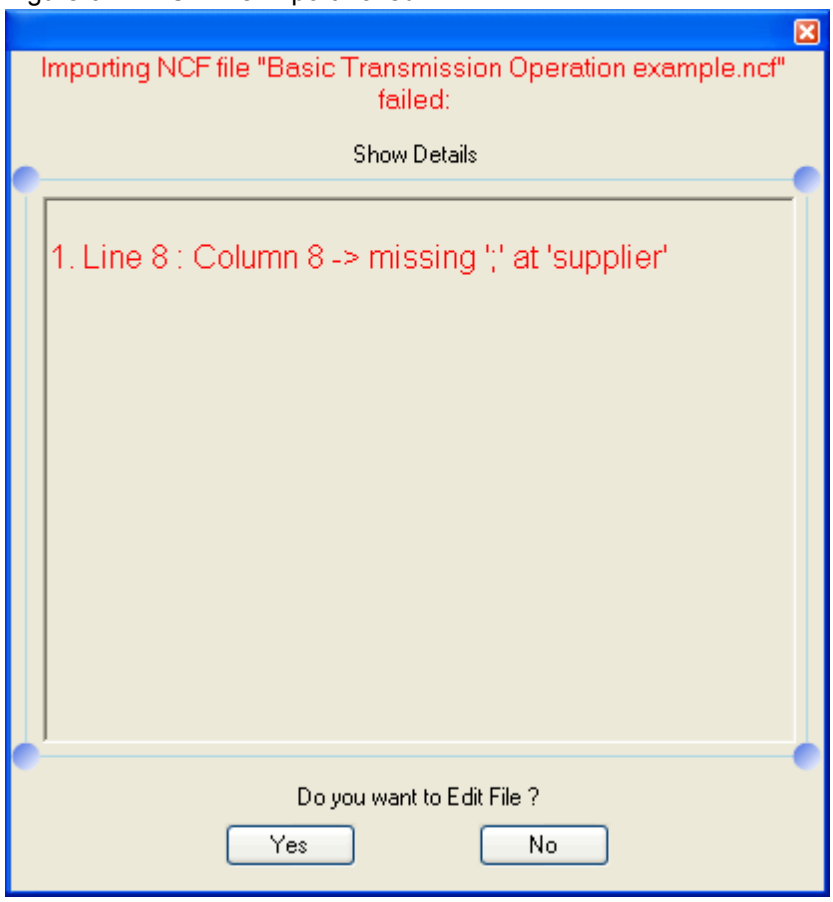
The syntax for *.ncf and *.ldf files is verified according to the LIN Node Capability Language Specification (Revision 2.1) and to the LIN Configuration Language Specification (Revision 2.1), respectively.

Figure 5. List of Available Nodes of NCF File to Import



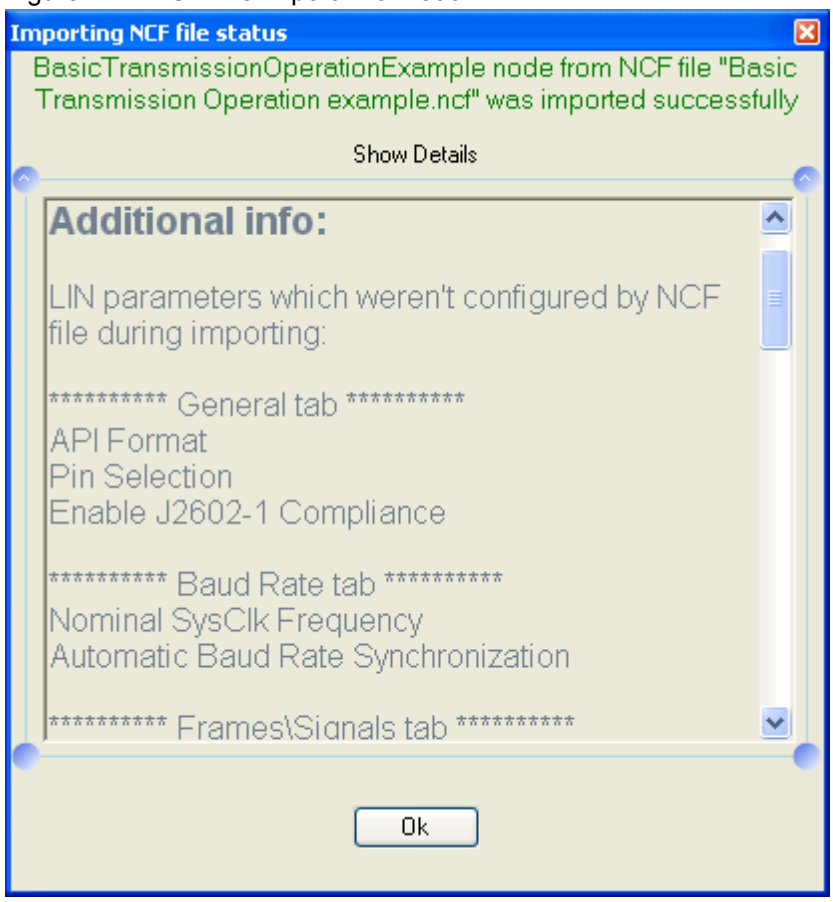
If the imported file contains errors, a dialog window similar to Figure 6 is displayed. There are two options in this case: edit the imported file to correct the errors using LIN Enhanced Editor Tool (see section “Tools Menu” for more information) or cancel the import by clicking the **No** button.

Figure 6. NCF File Import Failed



After the node to import is chosen and the import to the user module is completed, a dialog box that describes the importing results is displayed (see Figure 7). The importing results contain the LINS User Module parameters that were not affected during import.

Figure 7. NCF File Import Information



Export

File > Export menu (see Figure 8) enables you to save information about the user module configuration into a Node Capability File (NCF).

Figure 8. LINS User Module Wizard Export Menu

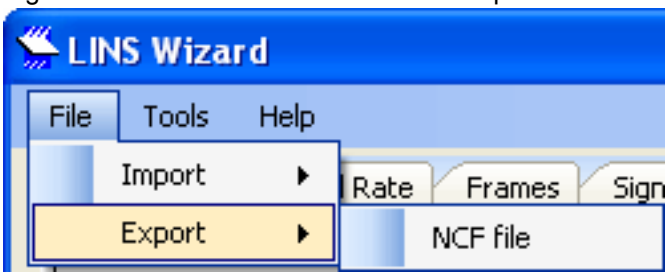
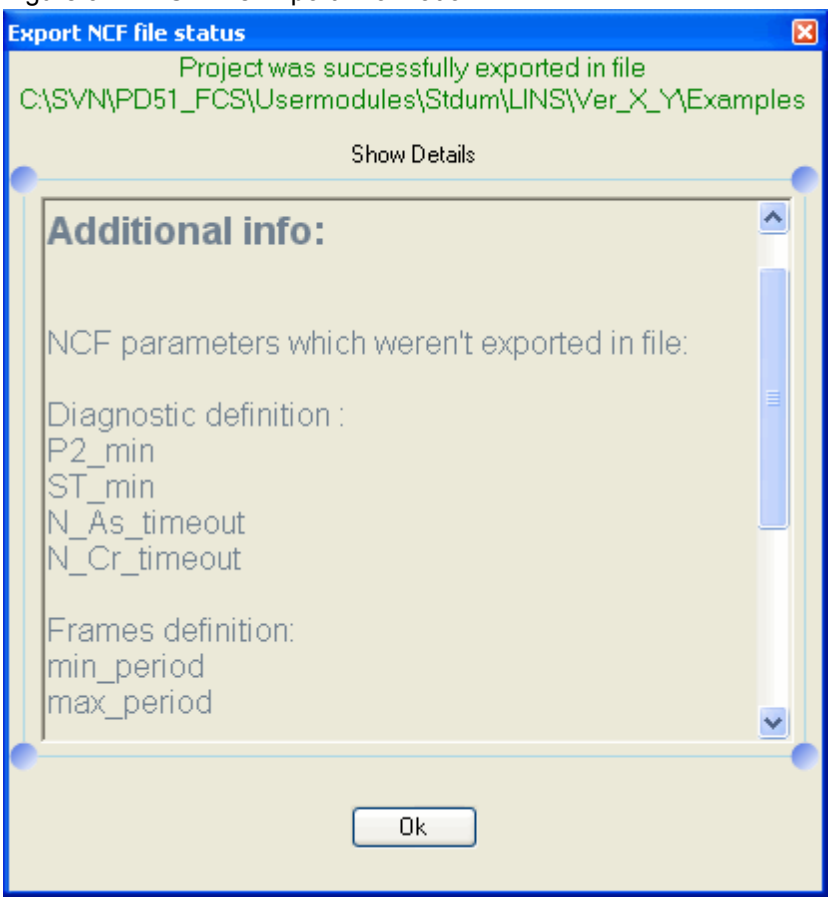


Figure 9. NCF File Export Information

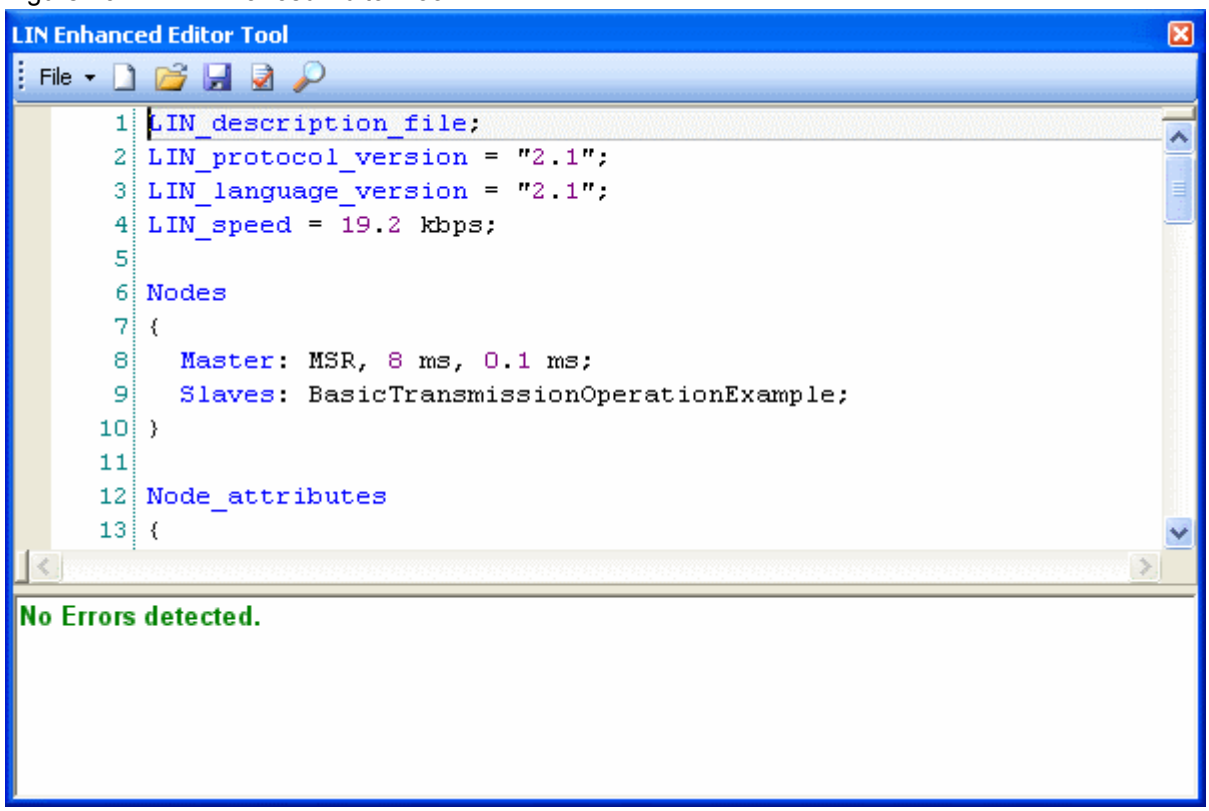


Tools Menu

Tools > LIN File Text Editor menu opens LIN Enhanced Editor Tool (Figure 10).

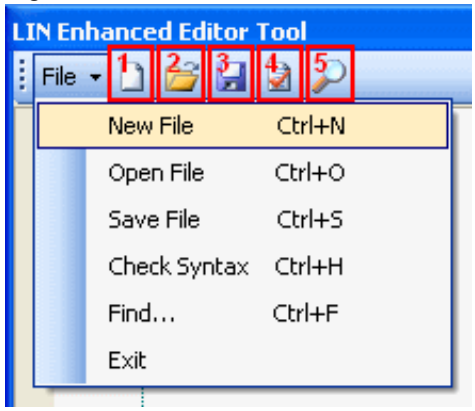
This tool is used to create, edit, and verify the syntax of the NCF/LDF file. The syntax for *.ncf files is verified according to LIN Node Capability Language Specification (Revision 2.1). The syntax for *.ldf files is verified accordingly to the LIN Configuration Language Specification (Revision 2.1).

Figure 10. LIN Enhanced Editor Tool



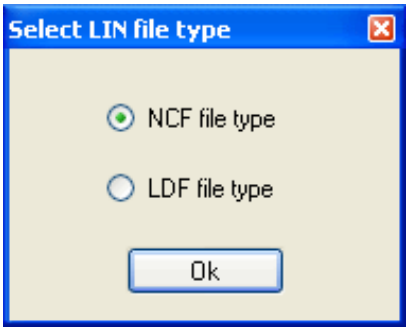
There is a toolbar at the top of “LIN Enhanced Editor Tool” window. The toolbar buttons functionality is described:

Figure 11. LIN File Text Editor Toolbar



1. “New File” Button

Creates a new file of the selected LIN file type.



2. “Open File” Button

Opens the specified existing LIN file.

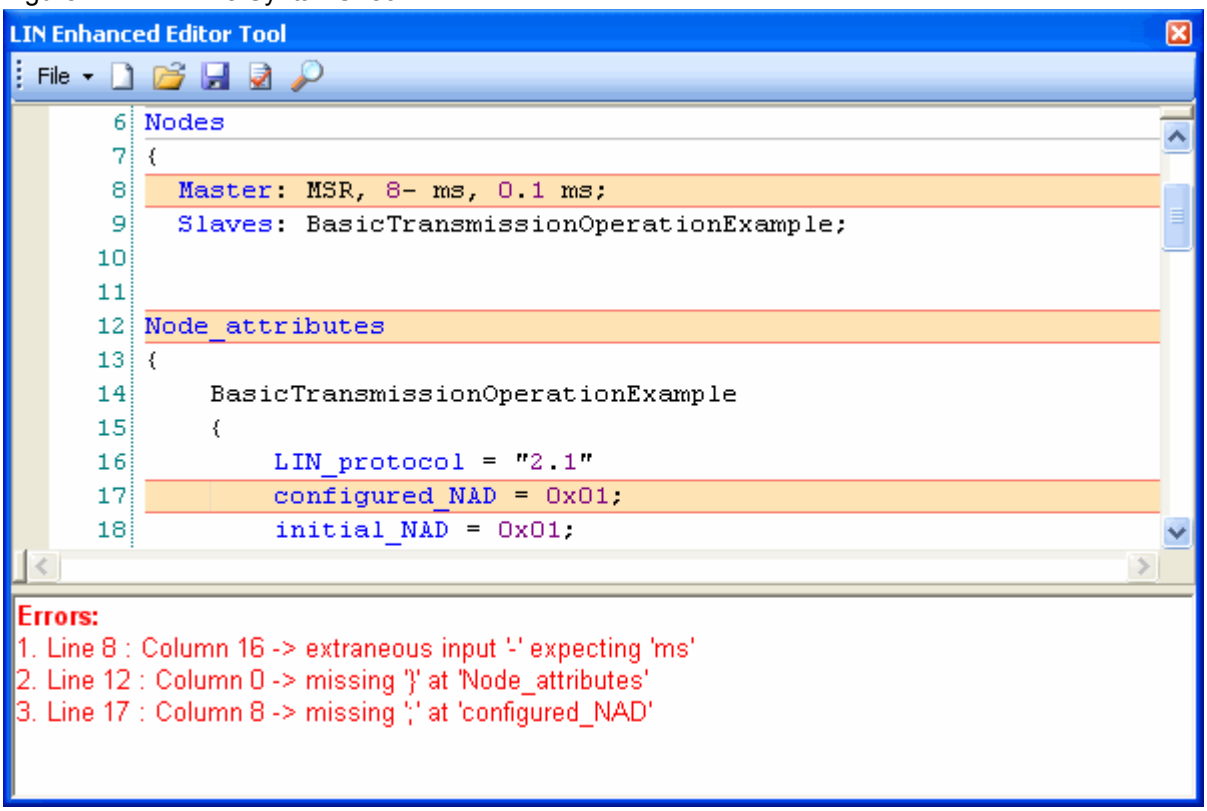
3. “Save File” Button

Saves the created LIN file to the specified location.

4. “Check Syntax” Button

This control enables 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 12). 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 12. LIN File Syntax Check



5. “Find” Button

This tool enables you to find the specified in the search field term in a LIN file (Figure 13). The “Find Next” button allocates the next match. If the “Mark Line” checkbox of the tool is enabled, then the lines containing the necessary term are labeled with yellow circles after clicking the “Find All” button. The “Style found token” checkbox enables or disables highlighting of the found token in yellow after clicking the “Find All” button, as shown in Figure 14. The “Clear” button removes all highlighted tokens.

Figure 13. Find Form

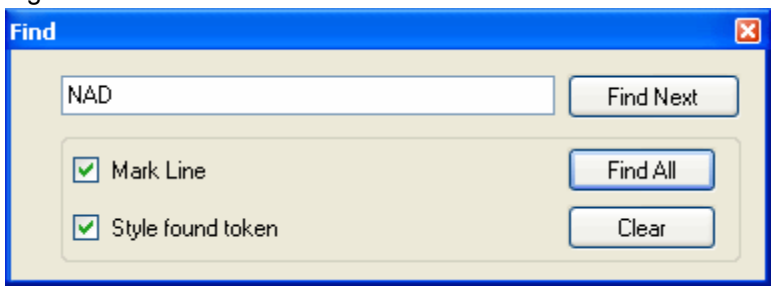
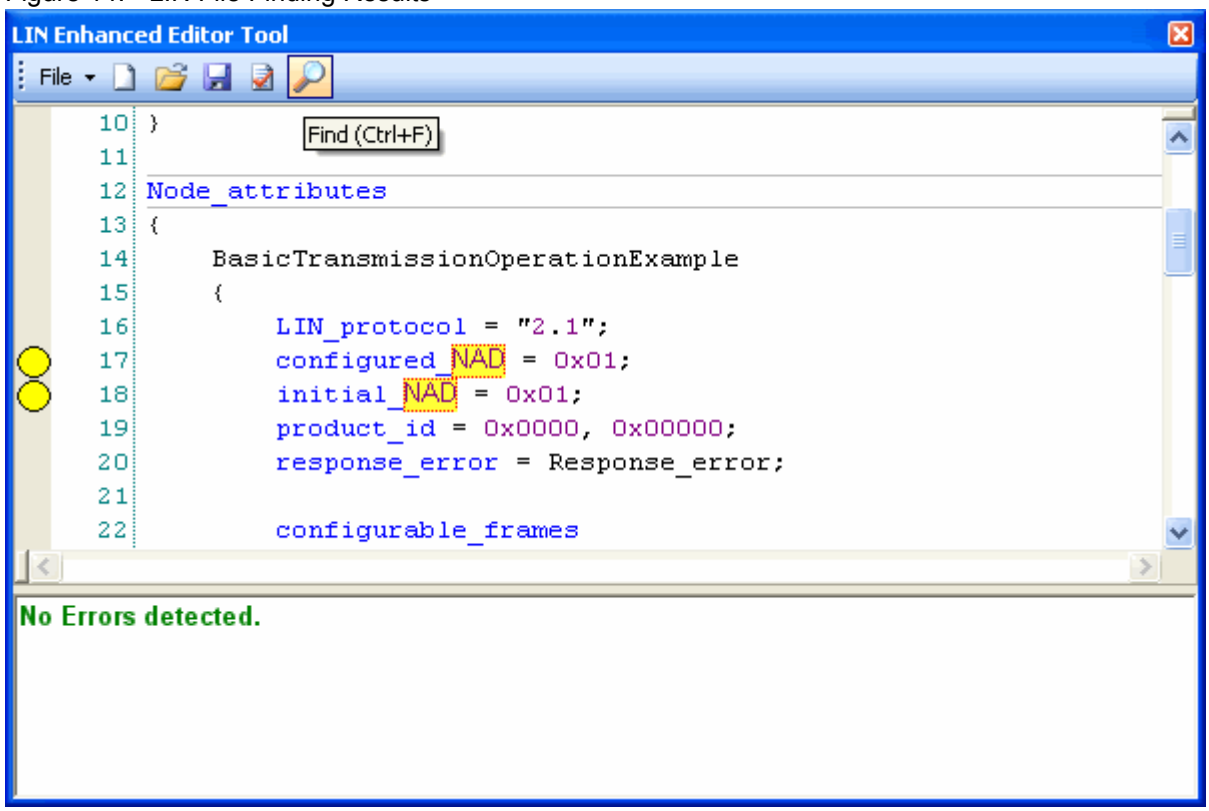


Figure 14. LIN File Finding Results



All tools are also available in File Menu of the “LIN Enhanced Editor Tool” and through the appropriate shortcuts.

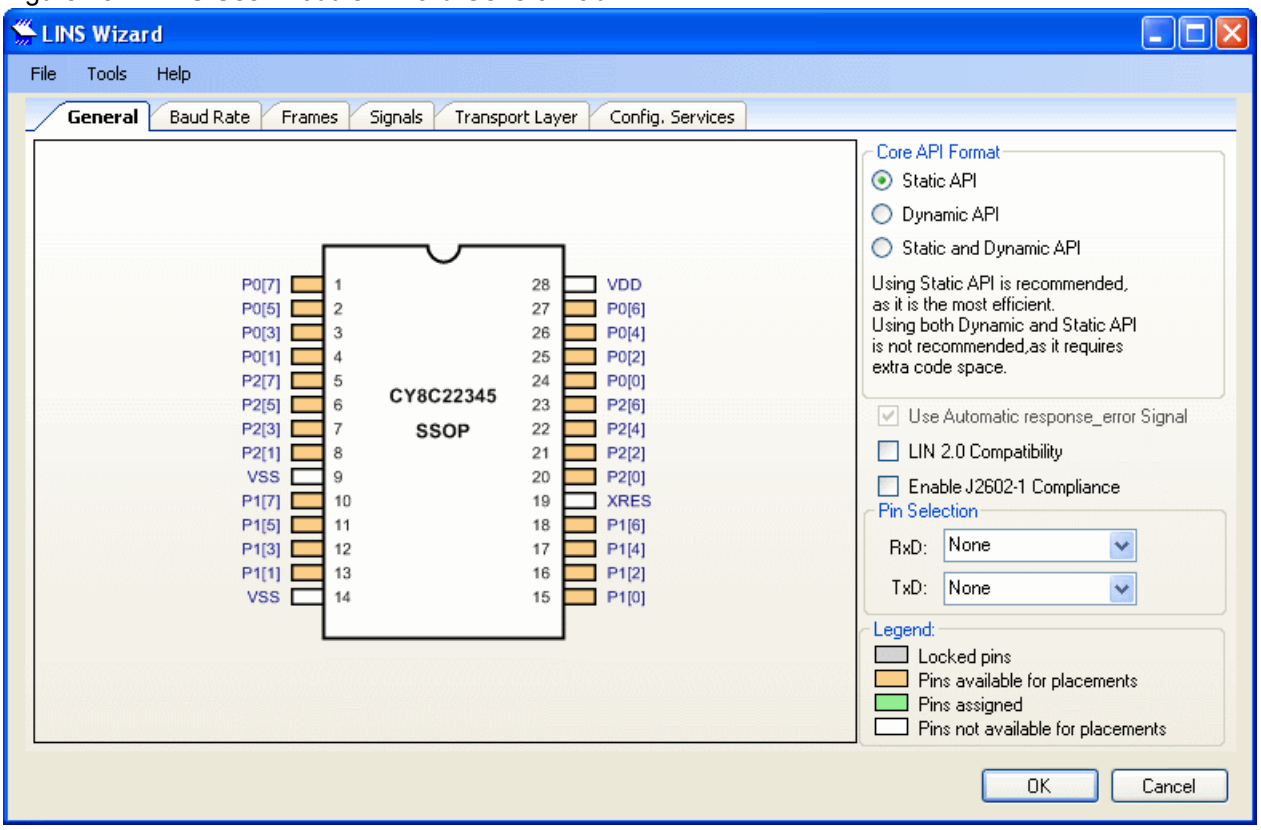
Help Menu

This menu calls the LINS User Module Wizard Help. It shortly describes the LINS User Module wizard main features.

General Tab

The LINS User Module Wizard General Tab is shown in Figure 15. This tab is used to configure the miscellaneous or general settings for the user module.

Figure 15. LINS User Module Wizard General Tab



Core API Format

This option allows selecting the LIN API format. The LIN specification defines two different API formats: Static and Dynamic.

If the “Static API” option is selected, then all signal, interface, or other names appear in the name of the API function. This means the set of available API functions (and the names of the API functions) changes as various settings of the user module are changed.

If you select the “Dynamic API” option, then all signal, interface, or other names will be passed into the function as a parameter. This means that there is one finite set of API functions that does not change as the user module settings are changed.

If you select the option for both “Static and Dynamic API”, then both API versions are implemented. Choosing this option is not recommended in most cases, because it requires a larger API function set which requires more flash memory use.

Pin Selection

This option enables you to select which pins are used for the user module RxD and TxD signals. See “PSoC and LIN Bus Hardware Interface” section for information on the external connections of the RxD and TxD signals.

If any GPIO is owned, claimed, or used by another user module, it is not shown in either of the Pin Selection dropdown boxes.

Always use the wizard to change or set the RxD and TxD signal pins.

Automatic Error Signal Selection

There is a “Use Automatic response_error Signal” checkbox on the tab. This box is always checked, so a 1-bit signal is automatically added in the Signals Tab of the wizard. This signal has a default name of “response_error”. This signal is set automatically by the user module whenever a "response error" occurs. The user module 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 user module should be compatible with the LIN 2.0 Specification. The status of this checkbox affects other areas of the wizard.

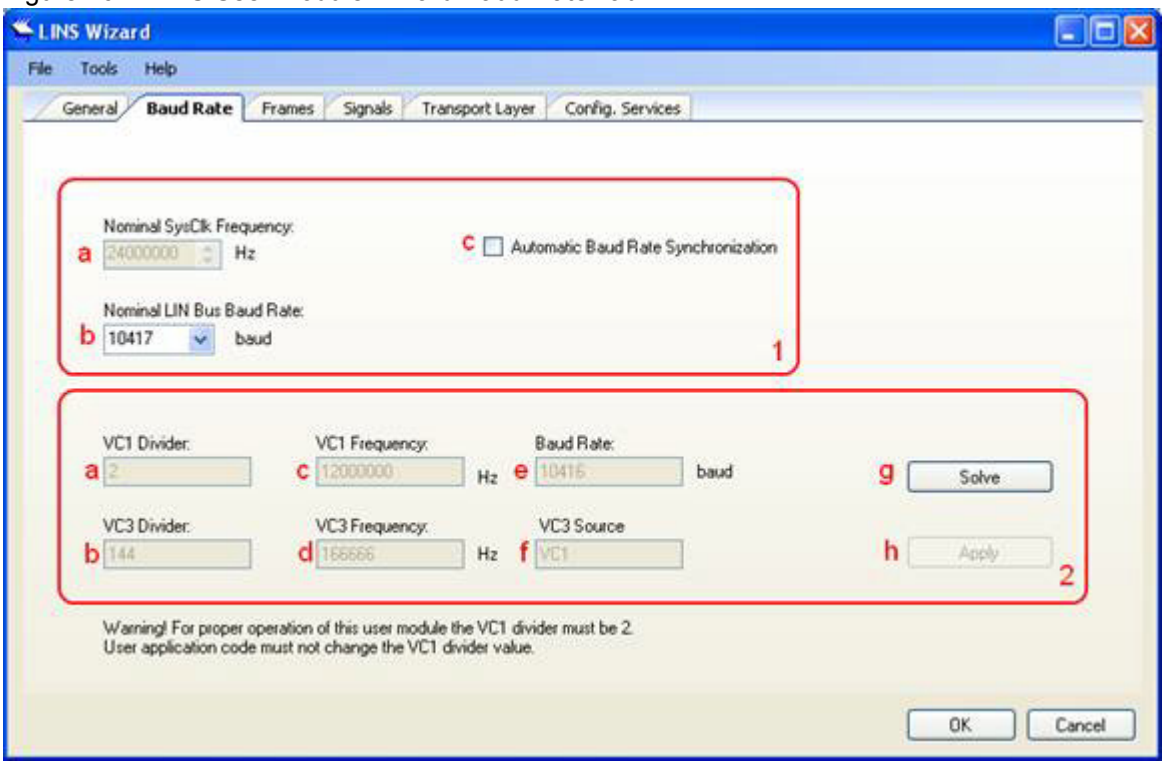
SAE J2602-1 Compliance

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 user module to be J2602-1 compliant. The status of this checkbox affects other areas of the wizard.

Baud Rate Tab

The Baud Rate Tab helps set up the system clock dividers (VCx clocks), based on the required LIN baud rate.

Figure 16. LINS User Module Wizard Baud Rate Tab



1 (a) Nominal SysClk Frequency

If you set the “SysClk Source” option to “Internal” (in the Global Resources panel of PSoC Designer), then this field is read-only. If you select the “External” SysClk option in the Global Resources parameters, this field is editable. In this case the field has a default value of “24000000”.

Note You must rerun the wizard (open the wizard and click “OK”) when you change the SysClk frequency or the external clock option of the Global Resource settings.

1 (b) Nominal LIN Bus Baud Rate

You must 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 wizard does not allow selecting baud rates outside of this range. The existing dropdown options for this box are 19200, 10417, 9600, and 2400. However, you can type in any value between 1000 and 20000.

1 (c) Automatic Baud Rate Synchronization

This option allows you to enable or disable “Automatic Baud Rate Synchronization”.

Generally, this option should be enabled unless a crystal or other very accurate external clock is used with the PSoC device.

If this option is enabled, the user module measures the exact baud rate of the bus from the sync byte field of each LIN frame header. It then sets the VC3 clock to match the baud rate of the bus to appropriately process the LIN frame. In this case, the VC3 clock frequency is changed at run-time. As a result, it is recommended that VC3 not be used to clock other resources other than this user module.

If this option is disabled, the user module does not measure the baud rate from the sync byte field. Instead, it just receives the sync byte field as a "0x55" data byte. In this case, the VC3 clock frequency is not changed at run-time by this user module, and therefore VC3 can be used to clock other resources.

As required by the LIN 2.1 Specification, LIN slave nodes with a frequency deviation of ±1.5% 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%, then the slave node must use Automatic Baud Rate Synchronization to measure the sync byte field of each frame.

Therefore, the frequency deviation specifications must be checked for the clock source that SysClk is derived from (this is typically the Internal Main Oscillator (IMO) in the PSoC). If the frequency deviation is ±1.5% or less, then this option should be disabled. If the frequency deviation is more than ±1.5%, then this option should be enabled.

2 (a-f) VC1/VC3 Divider, VC1/VC3 Frequency, VC3 Source, and Baud Rate boxes

These text boxes are always read-only. They display information about dividers and frequencies generated when you click the Solve button.

2 (g) Solve Button

There is a Solve button on the Baud Rate Tab of the wizard. It takes the inputs of the Nominal LIN Bus Baud Rate value and Nominal SysClk Frequency value and generates the output values in the output fields ((VC1, VC1 Frequency, VC3, VC3 Frequency, and Baud rate).

This button can be clicked multiple times. Each time it is clicked, the next set of valid values are displayed in the output boxes. This is useful when there are more than one valid combination of divider values that can be used to achieve the desired baud rate. Clicking this button multiple times enables you to select the exact combination of settings required.

2 (h) Apply Button

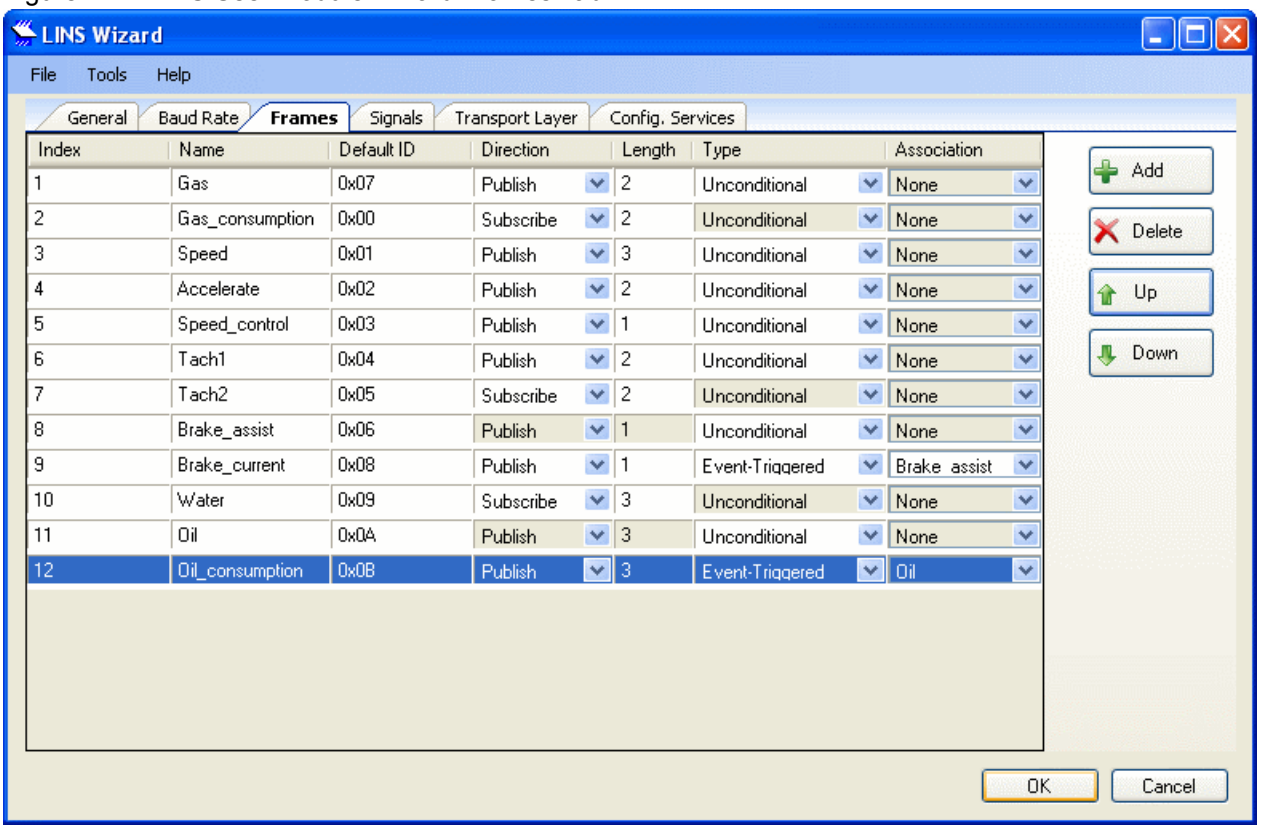
This button sets the Global Resources clock parameters (VC1, VC3, and VC3 Source) in a project.

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 user module 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 user module) must respond.

Figure 17. LINS User Module Wizard 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 with one LIN frame. Note that this tab only shows “user” LIN frames. The MRF and SRF frames are supported by this user module 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 for 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 frame IDs are dynamic. In other words, the LIN master has the ability to reconfigure frame IDs at run-time. A value from 0x00 to 0x3B must be entered into these cells. The values can be entered in hex or decimal format.

The “Message ID” column is not shown in Figure 17. This is because it is not normally visible. This column is only available if the LIN 2.0 Compatibility checkbox in the General Tab of the wizard has been checked. 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, then this user module 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. A value from 1 to 8, inclusive, must be entered in these fields.

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”. It is not possible to 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”, then the name of this frame must be changed 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 name(s) of any existing un-associated 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, then 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 it is associated with. Therefore, the name of an Event-Triggered frame appears only in unconditional frame rows in which these criteria apply. If the global “OK” button of the wizard is clicked, or if this tab is exited by clicking on another tab, the wizard checks to ensure that there are no Event-Triggered frames which 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 at this tab.

The “Add” button adds a new frame to the table.

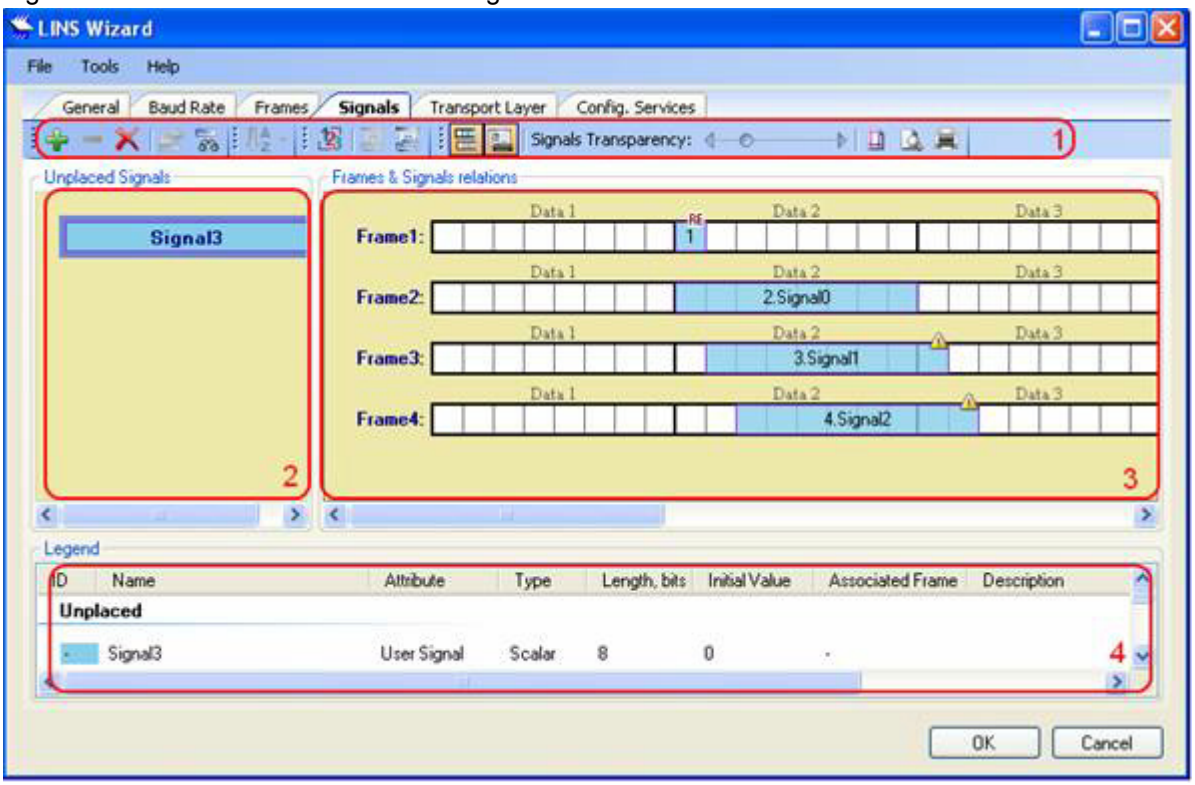
The “Delete” button deletes the currently selected frame from the table. The index number fields are changed accordingly. If a frame is deleted on this tab, any signals that are packed into it (configured with the Signals Tab) are moved into the "Unplaced Signals" region (See “Sort Signals” button in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

The "Up" and "Down" buttons can be used to reorder the Index number values for each frame.

Signals Tab

This tab of the wizard is used to define the "signals" that are packed into the LIN frames.

Figure 18. LINS User Module Wizard Signals Tab



Graphical Region

The “Frames & Signals relations” graphical region of this tab displays interactive graphics of the frames and the signals that you have defined with the wizard.

The "Unplaced Signals" 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 and 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.

Frame Graphics

There is one “Frame” graphic that represents each frame defined in the Frames Tab of the wizard. Note that the frame graphics are “static”; the graphics do not change while in this tab and you cannot interact directly with these graphics. The graphic for a frame appears as a string of consecutive bytes and bits. These bits are white when no signal occupies them. The graphic has n bytes and n*8 bits, where n is the number of data bytes the user has chosen for the frame in the Frames Tab. The name of each frame appears to the left of each frame graphic.

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 18). A signal is able to be placed on top of the frames using drag and drop methods. These signals occupy bits and/or bytes of the frames. The length of each bar is m bits long, where m is the number of bits you chose for that signal. Clicking on a signal “bar” graphic selects that signal. The index number of each signal is displayed on each signal bar graphic, which has already

been placed on the frame graphics. An index number is assigned to each signal in the order it is added. If a signal is deleted, all signal index numbers shift appropriately if there are any index number gaps.

Response_error Signal

The 1-bit response_error signal is automatically added in the Signals Tab of the wizard. It is possible to change the name of the response_error signal, but this signal cannot be deleted from the Signals Tab.

There may be only one instance of this signal and the name of the response_error signal must be unique for this user module. The response_error signal is a bool signal and can be placed anywhere on a frame which is published by the LIN slave.

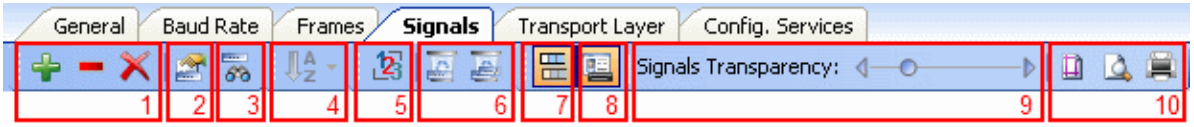
The purpose of this signal is to report status information to the LIN master.

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 Signals Tab. This toolbar provides an easy way to manage the signals on the tab.

Figure 19. Signals Toolbar



1. “Add/Delete” Buttons

The “Add Signal” button adds a signal to the "Unplaced Signals" region. The “Delete Signal” button removes selected signal from the user module. 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 enables 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 which correspond to Event-triggered frames in the "Frames&Signals relations" region.

8. “Show/Hide Legend” Button

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

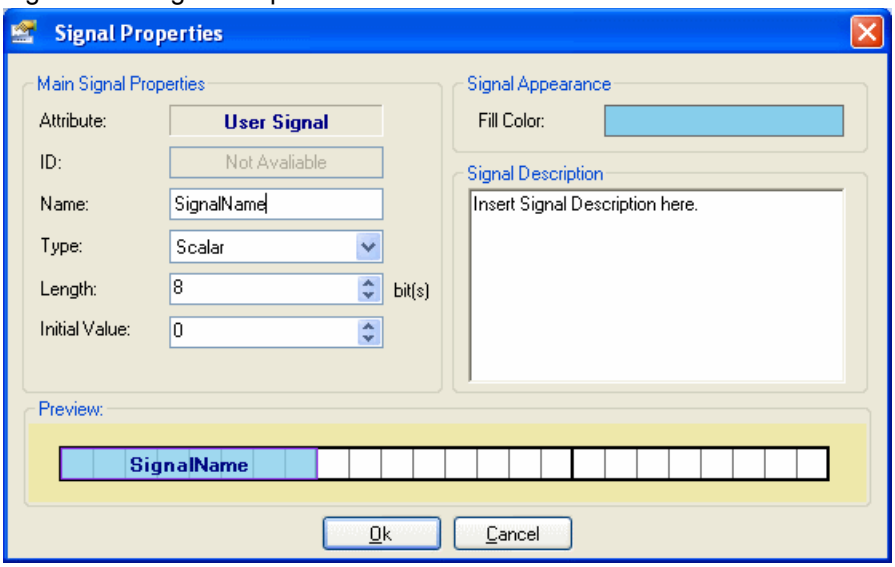
9. The "Signals Transparency" slider sets the transparency for signals graphics.

10. The “Print” buttons print out the “Frames & Signals relations” region.

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 20). 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 20. 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. A name must be entered for the signal that is 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.

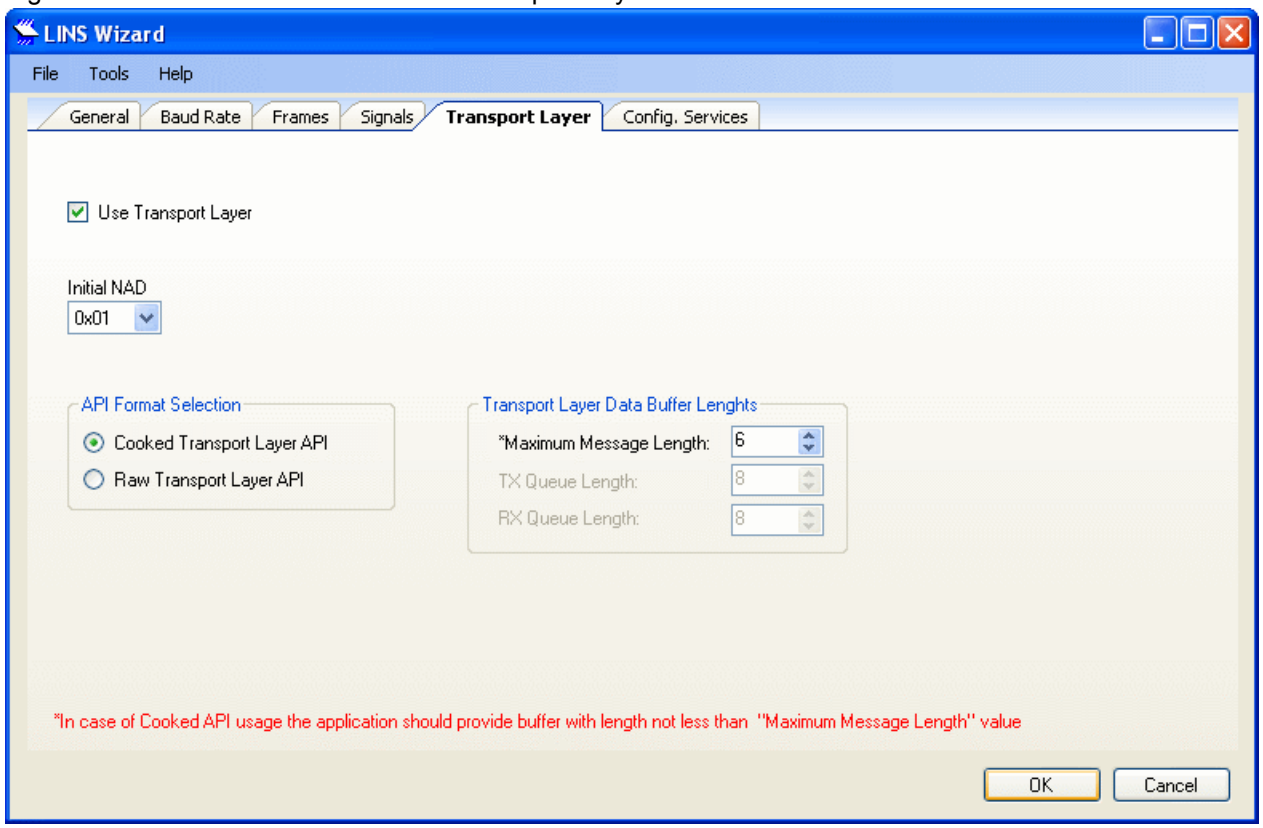
Preview

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 21.

Figure 21. LINS User Module Wizard Transport Layer Tab



Use Transport Layer Option

There is a "Use Transport Layer" checkbox on this tab. If this box is not checked, the user module will not support the Transport Layer. If it is checked, the user module supports the Transport Layer. See the LIN 2.1 Specification for detailed information on the Transport Layer.

Initial NAD Field

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 diagnostics services. The NAD value of 0x7F is reserved as a “wildcard” NAD. Therefore, the wizard restricts you from entering 0x00, 0x7E, or 0x7F into this field.

If the J2602-1 Compliance box is checked, the Initial NAD value on the Transport Layer Tab is restricted to 0x60 to 0x6F. The default value is 0x60. Initial Value range is further restricted based on the number of frames that are used on the Frames Tab of the wizard. See the following table 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 - 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

API Format Selection

This control is used to select the format for the Transport Layer API functions. There is a "Cooked" option and a "Raw" 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.

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, since there are up to 6 Transport Layer message data bytes in messages that use only one frame. This user module only supports Transport Layer messages with lengths up to 256 bytes. Note that the actual Transport Layer message buffer resides in the application code of the node.

TX Queue Length/RX Queue Length

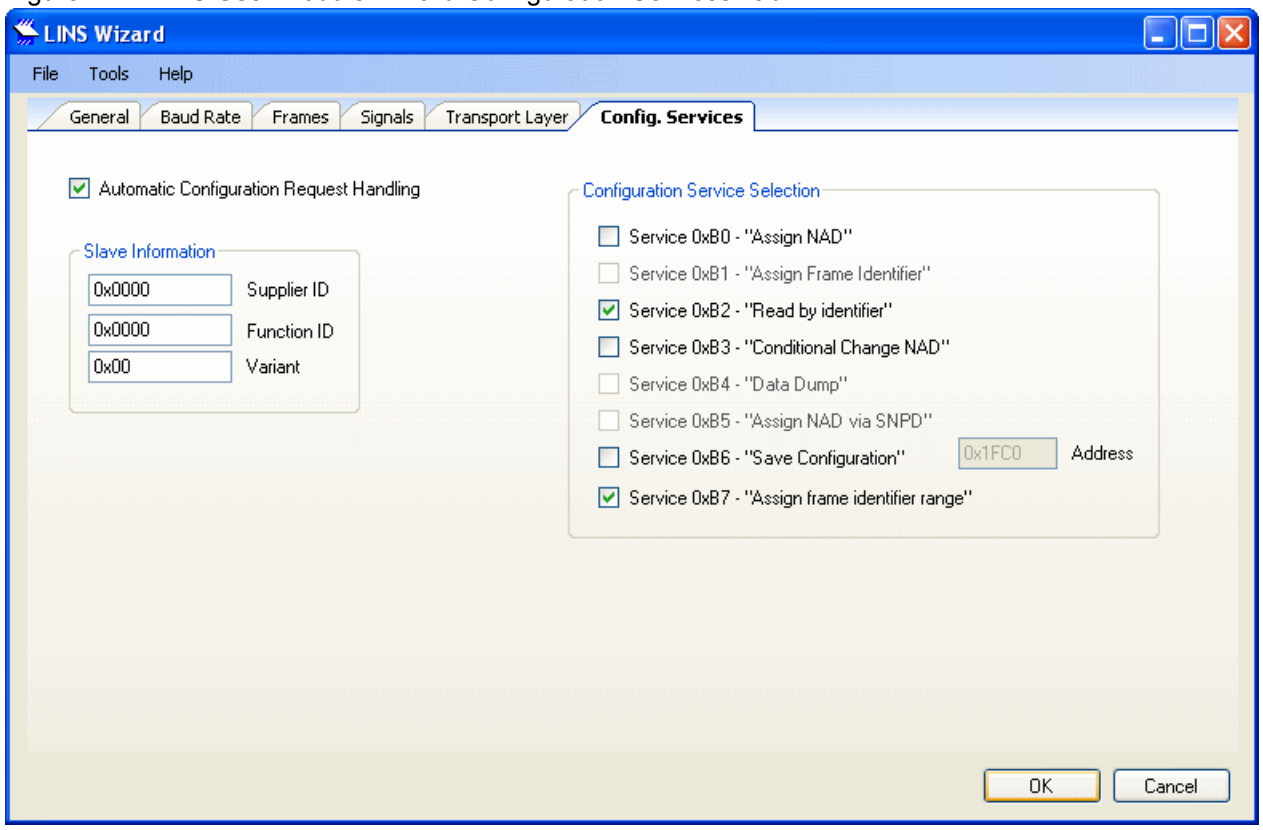
These properties are only applicable when the Raw API format is selected. When using the Raw API format, there is a message "queue" that buffers the frame response data that is being sent or received. If the slave cannot update the queues very quickly, then the queue lengths should be made longer. If the slave can update the queues very quickly, then the queues can be made shorter to decrease RAM use.

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 user module 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 user module supports some of them. There is 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 22. LINS User Module Wizard Configuration Services Tab



Automatic Configuration Request Handling

The user module is designed in such a way that it will automatically handle 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 facilitate this option, there is an “Automatic Configuration Request Handling” checkbox on this tab. If the box is checked, all of the other options on the tab are available. If the box is un-checked, then all of the other options on the tab are grayed out.

Any service that is enabled in this tab will be automatically handled by this user module. 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.

Slave Information

If you have checked the “Automatic Configuration Request Handling” checkbox, 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.

Configuration Service Selection

Each of the supported configuration service requests is listed on the tab with a checkbox for each one. You can individually select which 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.

Note that this service request is not likely to be needed for this user module, 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 checked on the General Tab of the wizard.

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

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 user module 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 user module only supports the "LIN Product Indentification" 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 (non-volatile) 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 user module.

Service 0xB5 – “Assign NAD via SNPD”

The "Assign NAD via SNPD" service is option in the LIN 2.1 specification and is not supported by this user module.

The "Targeted Reset" configuration service request is defined in the J2602-1 specification. Therefore, the 0xB5 service is only available (as "Targeted Reset") when the "Enable J2602-1 Compliance" checkbox is checked on the General Tab of the wizard. If a Targeted Reset request is processed by this slave, a flag is set in the LINS_bReadStatus API to let the application know that a Targeted Reset should occur.

Service 0xB6 – “Save Configuration”

This is an optional service request in the LIN 2.1 specification.

It is possible for the slave device to save its configuration data (NAD value and PID values) in non-volatile 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 non-volatile memory (flash).

For the application to overwrite the non-volatile configuration data, the data must be stored at a known location in flash. The flash address can be chosen in this tab of the wizard when this service is enabled. It is highly recommended to choose a flash address that is at the beginning of a flash block. The default flash block chosen for this data is the last flash block in the PSoC device. This is the recommended flash block in which the data should be stored.

The configuration data is stored in flash in the form of a C data structure. The members in this data structure are the NAD (1 byte), the PIDs (1 byte for each frame), the serial number (4 bytes, always 0x00000000), the Supplier ID (2 bytes), the Function ID (2 bytes), the Variant (1 byte), and the Message IDs (optional, 2 bytes for each frame). The ld_read_configuration and the ld_set_configuration API functions only read or set the NAD and PID values in this data structure. However, the information is all stored together in 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.

Parameters and Resources

Most properties and parameters of this user module are accessed through the user module wizard. However, the following user module properties are accessed in the Parameters window of PSoC Designer.

CPUClk Speedup

This parameter is used to automatically speed up the CPUClk to the highest allowed frequency any time any ISR of this user module is entered. The CPUClk setting is reset back to the normal setting just before any ISR of this user module exits.

This feature should only be enabled when the CPUClk normally does not operate at the maximum frequency. If the CPUClk is going to operate at the maximum allowed frequency, then this feature should be disabled.

Interrupt Generation Control

The following two parameters InterruptAPI and IntDispatchMode are only accessible by setting the Enable Interrupt Generation Control check box in PSoC Designer. This is available under **Project > Settings ... > Chip Editor**.

InterruptAPI

The InterruptAPI parameter allows conditional generation of a user module's interrupt handler and interrupt vector table entry. Select "Enable" to generate the interrupt handler and interrupt vector table entry. Select "Disable" to bypass the generation of the interrupt handler and interrupt vector table entry. If the Receive Command Buffer is to be used, then the InterruptAPI parameter should be set to "Enable". Properly selecting whether an Interrupt API is to be generated is recommended particularly with projects that have multiple overlays where a single block resource is used by the different overlays. By selecting Interrupt API generation only when it is necessary, the need to generate an interrupt dispatch code might be eliminated, thereby reducing overhead.

IntDispatchMode

The IntDispatchMode parameter is used to specify how an interrupt request is handled for interrupts shared by multiple user modules existing in the same block but in different overlays. Selecting "ActiveStatus" causes firmware to test which overlay is active before servicing the shared interrupt request. This test occurs every time the shared interrupt is requested. This adds latency and also produces a nondeterministic procedure of servicing shared interrupt requests, but does not require any RAM. Selecting "OffsetPreCalc" causes firmware to calculate the source of a shared interrupt request only when an overlay is initially loaded. This calculation decreases interrupt latency and produces a deterministic procedure for servicing shared interrupt requests, but at the expense of a byte of RAM.

Application Programming Interface

The Application Programming Interface (API) routines are provided as part of the user module to allow the designer to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the “include” files.

Only one instance of this user module can be placed in the project; this also applies to loadable configurations.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns “LINS_1” to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of some global function names, variables, and constant symbols. In the following descriptions the instance name has been shortened to LINS for simplicity.

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile” policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API function may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

The LINS User Module presents an API as defined in section 7 of the LIN 2.1 specification. All required API functions are listed here.

The LIN Application Program Interface Specification Revision 2.1 strictly defines the naming convention for API, data types, etc. Because of this, the standard Cypress user module naming conventions are used only for additional functions (such as LINS_bReadStatus) that are not required or defined by the LIN 2.1 specification.

Two versions exist of most of the API functions:

- Static calls embed the name of the LIN signal, LIN interface or LIN flag handle in the name of the call. The format for these calls is the following:
  - for interface handles: "[UM_API_function_name]_[UM_instance_name]"; for example, "l_ifc_ioctl_LINS_1" if the User Module instance name is "LINS_1"
  - for signal handles: "[UM_API_function_name]_[signal_name]"; for example, "l_u8_rd_Sig1" - for a signal named "Sig1"
  - for flag handles: "[UM_API_function_name]_[flag_name]"; for example, "l_flg_tst_Sig1" - for a flag that corresponds with a signal named "Sig1"; "l_flg_tst_Sig1Frame1" for a flag that corresponds with a duplicated signal named "Sig1" that is located in a frame named "Frame1".
- Dynamic calls provide the LIN signal or LIN interface as a parameter. The constants which should be passed as a parameter to dynamic versions of core APIs can be found in the user module header file. The format for these constants are:
  - for interface handles: "[UM_instance_name]_IFC_HANDLE"; for example, "LINS_1_IFC_HANDLE" if the user module instance name is "LINS_1"

  - for signal handles: "[signal_name]_SIGNAL_HANDLE"; for example, "Sig1_SIGNAL_HANDLE" for a signal named "Sig1"
  - for flag handles: "[flag_name]_FLAG_HANDLE"; for example, "Sig1_FLAG_HANDLE" for a flag that corresponds with a signal named "Sig1"; "Sig1Frame1_FLAG_HANDLE" for a flag that corresponds with a duplicated signal named "Sig1" that is located in a frame named "Frame1".

The LIN core defines the following data types:

- l_bool - 0 is false, and non-zero (>0) is true
- l_u8 - Unsigned 8 bit integer
- l_u16 - Unsigned 16 bit integer
- l_ioctl_op - Unsigned 8 bit integer
- l_irqmask - Unsigned 8 bit integer
- l_flag_handle - Unsigned 8 bit integer
- l_signal_handle - Unsigned 8 bit integer
- l_ifc_handle - Unsigned 8 bit integer

Note

1. The Core API Functions and the Transport Layer API Functions (if used) are dynamically generated by the Wizard depending on wizard settings.
2. Signal packing/unpacking is implemented more efficiently in software based nodes if signals are byte aligned and/or if they do not cross byte boundaries.
3. Static Signal Interaction functions are generated depending on the locations of the signals in a frame.

Global Variables

For each frame that this user module has defined (including the MRF frame and the SRF frame), there is an array of bytes for this frame that has the length equal to the frame length. Each array for each of the frames is global. This way, you can access these arrays directly if you want to. This potentially saves a significant amount of RAM if you do not use the API to access these global arrays.

The LINS_wInactivityVar global variable is the count of timer overflows needed to detect bus inactivity.

If Transport Layer is enabled then:

- LINS slave node configuration data is available through the LINSlaveConfiguration global structure, which provides the following information:
  - Initial NAD
  - PID Table
  - Serial Number of LIN slave
  - SupplierID
  - FunctionID
  - Variant
  - MessageID Table (if applicable)

- The LINS_wGlobal_TL_Timeout global variable is the count of timer overflows needed to detect time-out between packets reception/transmission in a multi PDU message (this variable is used when the Cooked Transport Layer API is used).

API Reference

LINS User Module API Group	LINS User Module API Subgroup	LINS User Module API Function
Core API Functions	Initialization	l_sys_init
	Signal Interaction Functions*	l_bool_rd
		l_u8_rd
		l_u16_rd
		l_bytes_rd
		l_bool_wr
		l_u8_wr
		l_u16_wr
		l_bytes_wr
	Notification Functions*	l_flg_tst
		l_flg_clr
	Interface Management Functions*	l_ifc_init
		l_ifc_wake_up
		l_ifc_ioctl
		l_ifc_rx
		l_ifc_tx
		l_ifc_aux
		l_ifc_read_status
	User-provided Call-outs	l_sys_irq_disable
		l_sys_irq_restore
Node Configuration Functions		ld_read_configuration
		ld_set_configuration

LINS User Module API Group	LINS User Module API Subgroup	LINS User Module API Function
Transport Layer Functions	Initialization	ld_init
	Raw Transport Layer API Functions	ld_put_raw
		ld_get_raw
		ld_raw_tx_status
		ld_raw_rx_status
	Cooked Transport Layer API Functions	ld_send_message
		ld_receive_message
		ld_tx_status
		ld_rx_status
Non-LIN-Specified API		LINS_bReadStatus

*For this user module API subgroups dynamic and static API versions exist. The name of the static API embeds signal, flag, or interface name depending on API subgroup. Refer to the static and dynamic calls format description for more informations.

Core API Functions

Initialization

l_sys_init

Description:

This function provides the initialization of the LINS User Module (copies all available PIDs in user module from ROM to RAM array).

The call to the l_sys_init is the first call you must use in the LINS User Module before using any other API function.

C Prototype:

```
l_bool  l_sys_init(void)
```

Assembly:

```
lcall  l_sys_init
```

Parameters:

None

Return Value:

0 - if the initialization succeeds

Non-zero value - if the initialization fails

Side Effects:

The A and X registers may be altered by this function.

Signal Interaction Functions

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

Signal Types

The signals can be of three different types:

Signal Type	Description
l_bool	1-bit signals; zero if false, non-zero otherwise
l_u8	for signals of the size 2 - 8 bits
l_u16	for signals of the size 9 - 16 bits

l_bool_rd

Description:

Reads and returns the current value of the signal. If an invalid signal handle is passed into the function, no action is done.

Static C Prototype:

```
l_bool  l_bool_rd_sss(void)
```

Dynamic C Prototype:

```
l_bool  l_bool_rd(l_signal_handle sss)
```

Static Assembly:

```
lcall  l_bool_rd_sss
```

Dynamic Assembly:

```
mov   A, sss
lcall  l_bool_rd
```

Parameters:

sss - signal handle of the signal to read.

Return Value:

Returns current value of signal.

Side Effects:

The A and X registers may be altered by this function.

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 done.

Static C Prototype:

```
l_u8  l_u8_rd_sss(void)
```

Dynamic C Prototype:

```
l_u8  l_u8_rd(l_signal_handle sss)
```

Static Assembly:

```
lcall  l_u8_rd_sss
```

Dynamic Assembly:

```
mov   A, sss
lcall  l_u8_rd
```

Parameters:

sss - signal handle of the signal to read.

Return Value:

Returns current value of signal.

Side Effects:

The A and X registers may be altered by this function.

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 done.

Static C Prototype:

```
l_u16  l_u16_rd_sss(void)
```

Dynamic C Prototype:

```
l_u16  l_u16_rd(l_signal_handle sss)
```

Static Assembly:

```
lcall  l_u16_rd_sss
```

Dynamic Assembly:

```
mov   A, sss
lcall  l_u16_rd_sss
```

Parameters:

sss - signal handle of the signal to read.

Return Value:

Returns current value of signal.

Side Effects:

The A and X registers may be altered by this function.

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 then an accidental data is read.

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

Assume that a byte array is 8 bytes long, numbered 0 to 7. Reading bytes from 2 to 6 from 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 C Prototype:

```
void  l_bytes_rd_sss(l_u8 start, l_u8 count, l_u8* const data)
```

Dynamic C Prototype:

```
void  l_bytes_rd(l_signal_handle sss, l_u8 start, l_u8 count, l_u8* const data)
```

Static Assembly:

```
mov   A, >data ; Load MSB destination address. Data will be stored to array whith this
address
push  A
mov   A, <data ; Load LSB destination address. Data will be stored to array whith this
address
push  A
mov   A, count ; Load Number of bytes to be read
push  A
mov   A, start ; Load data offset for reading
push  A
lcall  l_bytes_rd_sss
```

Dynamic Assembly:

```
mov   A, >data ; Load MSB destination address. Data will be stored to array whith this
address
push  A
mov   A, <data ; Load LSB destination address. Data will be stored to array whith this
address
push  A
mov   A, count ; Load Number of bytes to be read
push  A
mov   A, start ; Load data offset for reading
push  A
mov   A, sss
push  A
lcall  l_bytes_rd
```

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:

The A and X registers may be altered by this function.

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 done.

Static C Prototype:

```
void  l_bool_wr_sss(l_bool v)
```

Dynamic C Prototype:

```
void  l_bool_wr(l_signal_handle sss, l_bool v)
```

Static Assembly:

```
mov   A, v
lcall  l_bool_wr_sss
```

Dynamic Assembly:

```
mov   A, sss
mov   X, v
lcall  l_bool_wr
```

Parameters:

sss - signal handle of the signal to write;

v - value of the signal to be set.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_u8_wr

Description:

Writes the value “v” to the signal. If an invalid signal handle is passed into the function, no action is done.

Static C Prototype:

```
void  l_u8_wr_sss(l_u8 v)
```

Dynamic C Prototype:

```
void  l_u8_wr(l_signal_handle sss, l_u8 v)
```

Static Assembly:

```
mov   A, v
lcall  l_u8_wr_sss
```

Dynamic Assembly:

```
mov   A, sss
mov   X, v
lcall  l_u8_wr
```

Parameters:

sss - signal handle of the signal to write

v - value of the signal to be set

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_u16_wr

Description:

Writes the value “v” to the signal. If an invalid signal handle is passed into the function, no action is done.

Static C Prototype:

```
void  l_u16_wr_sss(l_u16 v)
```

Dynamic C Prototype:

```
void  l_u16_wr(l_signal_handle sss, l_u16 v)
```

Static Assembly:

```
; for v Binary:
mov   X, >v ; upper byte
mov   A, <v ; lower byte
lcall  l_u16_wr_sss
;or
mov   X, [v]          ; upper byte
mov   A, [v+1]        ; lower byte
lcall  l_u16_wr_sss
```

Dynamic Assembly:

```
; for v Binary:
mov   A, >v ; upper byte
push  A
mov   A, <v ; lower byte
push  A
mov   A, sss
push  A
call  l_u16_wr
;or
mov   A, [v] ; upper byte
push  A
mov   A, [v+1] ; lower byte
push  A
mov   A, sss
push  A
call  l_u16_wr
```

Parameters:

sss - signal handle of the signal to write;

v - value of the signal to be set.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

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 runtime. Note that when the sum of “start” and “count” is greater than the length of the signal byte array then an accidental memory area is to be affected.

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

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 byte 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 C Prototype:

```
void  l_bytes_wr_sss(l_u8 start, l_u8 count, const l_u8* const data)
```

Dynamic C Prototype:

```
void  l_bytes_wr(l_signal_handle sss, l_u8 start, l_u8 count, const l_u8* const data)
```

Static Assembly:

```
mov   A, >data ; Load MSB destination address. Data will be stored to array whith this
address
push  A
mov   A, <data ; Load LSB destination address. Data will be stored to array whith this
address
push  A
mov   A, count ; Load Number of bytes to be read
push  A
mov   A, start ; Load data offset for reading
push  A
lcall  l_bytes_wr_sss
```

Dynamic Assembly:

```
mov   A, >data ; Load MSB destination address. Data will be stored to array whith this
address
push  A
mov   A, <data ; Load LSB destination address. Data will be stored to array whith this
address
push  A
mov   A, count ; Load Number of bytes to be read
push  A
mov   A, start ; Load data offset for reading
push  A
mov   A, sss
push  A
lcall  l_bytes_wr
```

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:

The A and X registers may be altered by this function.

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

Notification flags are used to synchronize the application program with the LIN core. The flags will be 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 user module when the corresponding signal or frame is successfully sent or received.

In all flag API routines below the “fff” is the name of the flag, for example, l_flg_tst_RxEngineSpeed(). For dynamic flag API routines below the "fff" is a signal handle as defined above in the Application Programming Interface section.

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 C Prototype:

```
l_bool  l_flg_tst_fff(void)
```

Dynamic C Prototype:

```
l_bool  l_flg_tst(l_flag_handle fff)
```

Static Assembly:

```
lcall  l_flg_tst_fff
```

Dynamic Assembly:

```
mov   A, fff
lcall  l_flg_tst
```

Parameters:

fff - is the name of the flag handle.

Return Value:

Returns a C boolean indicating the current state of the flag specified by the name “fff”.

False - if the flag is cleared;

True - if the flag is not cleared.

Side Effects:

The A and X registers may be altered by this function.

l_flg_clr

Description:

Clears flag which 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 user module does not automatically clear notification flags. This routine is the only way that a notification flag can be cleared.

Static C Prototype:

```
void  l_flg_clr_fff(void)
```

Dynamic C Prototype:

```
void  l_flg_clr(l_flag_handle fff)
```

Static Assembly:

```
lcall  l_flg_clr_fff
```

Dynamic Assembly:

```
mov   A, fff
lcall  l_flg_clr
```

Parameters:

fff - is the name of the flag handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

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 user module, the interface name is the same as the user module instance name. This user module 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 LINS User Module instance which is specified by the name “iii”, it sets up internal functions such as the baud rate, starts up digital blocks which are used by the LINS User Module. This is the first call that must be performed, before using any other interface related LINS API functions.

Static C Prototype:

```
l_bool  l_ifc_init_iii(void)
```

Dynamic C Prototype:

```
l_bool  l_ifc_init(l_ifc_handle iii)
```

Static Assembly:

```
lcall  l_ifc_init_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_init
```

Parameters:

iii - is the name of the interface handle.

Return Value:

The function returns zero if the initialization was successful and non-zero if failed.

Side Effects:

The A and X registers may be altered by this function.

l_ifc_wake_up

Description:

The function transmits one wake up signal. The wake up signal is transmitted directly when this function is called. When you call this API function then application is blocked until a wake up signal is being transmitted on the LIN bus. The timing source for the l_ifc_wake_up API is CPUClk. The wake up delay time is automatically calculated and is independent from the user selected CPU Clock frequency.

Static C Prototype:

```
void  l_ifc_wake_up_iii(void)
```

Dynamic C Prototype:

```
void  l_ifc_wake_up(l_ifc_handle iii)
```

Static Assembly:

```
lcall  l_ifc_wake_up_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_wake_up
```

Parameters:

iii - is the name of the interface handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_ifc_ioctl

Description:

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

This function supports two operations for this user module.

1. The first operation is **Set Baud Rate** (L_OP_SET_BAUD_RATE operation code).

This operation changes the nominal baud rate that the user module operates at. Changing the baud rate requires VC3 (and possibly VC1) modification. Changing the baud rate requires a new BUS_INACTIVITY threshold value, so you have manually recalculate it and pass it to the l_ifc_ioctl API function as the third parameter.

The initial value of the BUS_INACTIVITY constant (which is the count of timer overflows needed to detect bus inactivity) is calculated by the user module wizard for a 7 second bus inactivity threshold (so “User selected Inactivity time” is 7 seconds by default).

The equation for BUS_INACTIVITY calculation is:

$$\text{Threshold}_{\text{INACTIVITY}} = \frac{8 \times F_{BAUD}}{256} \times \text{User_Selected_Inactivity_Time}$$

Equation 6

In Equation 6:

$\text{Threshold}_{\text{INACTIVITY}}$ – is a value that you should pass to the l_ifc_ioctl API function as the third parameter.

F_{BAUD} – is value of current LINS baud rate. Available range is from 1 to 20 kbaud.

User_Selected_Inactivity_Time – is time value in seconds until BUS_INACTIVITY flag is set if the slave detects no transitions on the bus. Recommended time range is from 4 to 10 seconds.

256 – is a Timer period value.

Global_TL_Timeout constant is the count of timer overflows needed to detect timeout between packets reception/transmission in the multi PDU message.

Global_TL_Timeout is available if Transport Layer is enabled and Cooked API functions are used.

The equation for Global_TL_Timeout calculation is:

$$\text{Threshold}_{\text{TL_TIMEOUT}} = \frac{8 \times F_{BAUD}}{256} \times \text{TL_Timing_Timeout}$$

Equation 7

In Equation 7:

$\text{Threshold}_{\text{TL_Timeout}}$ – is value that you must write to the LINS_wGlobal_TL_Timeout global variable after l_ifc_ioctl API function is called.

F_{BAUD} – is value of current LINS baud rate. Available range is from 1 to 20 kbaud.

TL_Timing_Timeout – is recommended time delay value which is equal to 1 sec (defined in the LIN 2.1 specification, section 3.2.5).

256 – is a Timer period value.

2. The second feature is **Sleep operation** (L_OP_SLEEP operation code).

The Sleep operation simply puts the LIN interface into a state so that the PSoC can be woken up by a dominant pulse on the bus. This operation puts the user module into a state where a falling edge on the bus generates an interrupt which wakes the PSoC up.

Static C Prototype:

```
l_u16  l_ifc_ioctl_iii(l_ioctl_op op, void* pv)
```

Dynamic C Prototype:

```
l_u16  l_ifc_ioctl(l_ifc_handle iii, l_ioctl_op op, void* pv)
```

Static Assembly:

```
mov   A, >pv ; Load MSB of pointer
push  A
mov   A, <pv ; Load LSB of pointer
push  A
mov   A, op ; Load operation code
push  A
lcall  l_ifc_ioctl_iii
```

Dynamic Assembly:

```
mov   A, >pv ; Load MSB of pointer
push  A
mov   A, <pv ; Load LSB of pointer
push  A
mov   A, op ; Load operation code
push  A
mov   A, iii
push  A
lcall  l_ifc_ioctl
```

Parameters:

The “iii” is the name of the interface handle to which the operation defined in “op” will be applied.

The “op” parameter is used to specify the operation.

The “pv” parameter is a 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 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_OP_SET_BAUD_RATE	0	BYTE, BYTE, WORD	Allows changing the baud rate
L_OP_SLEEP	1	None	Puts the LIN slave into Sleep state

For the “Set Baud Rate” operation, the “pv” parameter is a pointer to three consecutive parameter variables.

The first parameter is a byte that is the VC3 divider value to be applied. Note that when using the automatic baud rate synchronization feature, you must ensure to not set a VC3 divider value that is less than 100 or more than 222. If it is less than 100, the automatic baud rate synchronization function will not have enough resolution. If it is more than 222, then it is possible that the automatic baud rate synchronization function may fail because the value may overflow.

The second parameter is a byte that is the VC1 divider value to be applied. If the second parameter is an invalid value (that is, not from 2 to 16, inclusive), then the VC1 divider is not to be used in the clock chain, and VC3's source must be SysClk. If the second parameter's value is valid (that is, from 2 to 16, inclusive), then VC3's source is to be VC1, and VC1 must be set accordingly. The '0' value is not valid for the VC1 divider because dividing by zero is not possible. The '1' value is not valid for the VC1 divider value because the VC1 divider must never be set to divide by 1.

The BUS_INACTIVITY threshold value must be recalculated manually and passed to the l_ifc_ioctl API function as the third parameter. During the calculation, the recommended time range for User_Selected_Inactivity_Time value is from 4 to 10 seconds. The default value of User_Selected_Inactivity_Time is 7 seconds.

The following tables show example parameter values for the "Set Baud Rate" operation of this API function.

Baud Rate	$T_{INACTIVITY}$	First Parameter (BYTE)	Second Parameter (BYTE)	Third Parameter (WORD)
19200	7 s	156	255	4200
10417	7 s	144	2	2279
9600	7 s	156	2	2100
2400	7 s	156	8	525

Baud Rate	$T_{INACTIVITY}$	First Parameter (BYTE)	Second Parameter (BYTE)	Third Parameter (WORD)
19200	2 s	156	255	1200
10417	2 s	144	2	652
9600	2 s	156	2	600
2400	2 s	156	8	150

Return Value:

Symbolic Name	Value	Description
L_SUCCESSFUL_OP	0	An operation succeed
L_INVALID_OP	1	An invalid operation parameter is passed to the function

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

Side Effects:

The A and X registers may be altered by this function.

l_ifc_rx

Description:

This user module takes care of calling 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 C Prototype:

```
void  l_ifc_rx_iii(void)
```

Dynamic C Prototype:

```
void  l_ifc_rx(l_ifc_handle iii)
```

Static Assembly:

```
clall  l_ifc_rx_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_rx
```

Parameters:

iii - is the name of the interface handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_ifc_tx

Description:

This user module takes care of calling 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 C Prototype:

```
void  l_ifc_tx_iii(void)
```

Dynamic C Prototype:

```
void  l_ifc_tx(l_ifc_handle iii)
```

Static Assembly:

```
lcall  l_ifc_tx_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_tx
```

Parameters:

iii - is the name of the interface handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_ifc_aux

Description:

This user module takes care of calling 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 C Prototype:

```
void  l_ifc_aux_iii(void)
```

Dynamic C Prototype:

```
void  l_ifc_aux(l_ifc_handle iii)
```

Static Assembly:

```
lcall  l_ifc_aux_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_aux
```

Parameters:

iii - is the name of the interface handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

l_ifc_read_status

Description:

This function will return the status of the previous communication. Refer to the LIN 2.1 specification for detailed information on each status information field in the LINS status word.

Static C Prototype:

```
l_u16  l_ifc_read_status_iii(void)
```

Dynamic C Prototype:

```
l_u16  l_ifc_read_status(l_ifc_handle iii)
```

Static Assembly:

```
lcall  l_ifc_read_status_iii
```

Dynamic Assembly:

```
mov   A, iii
lcall  l_ifc_read_status
```

Parameters:

iii - is the 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 trig-gered frame collision	Bus activity	Go to sleep	Overrun	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 will be cleared after API is called.

Side Effects:

The A and X registers may be altered by this function.

User-provided Call-outs

l_sys_irq_disable

Description:

This function disables all interrupts for the user module. It returns a mask of the state that the interrupt mask bits were in. This function is essentially equivalent to the DisableInt API of most user modules. However, 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 great care be taken when using this API routine. It is likely that LIN communication failures will occur if the interrupts for this user module are disabled for too long.

This routine is supposed to be provided by the application. However, the LINS User Module implements this routine automatically. You can modify the code in the routine if necessary.

Static C Prototype:

```
l_irqmask  l_sys_irq_disable(void)
```

Static Assembly:

```
lcall  l_sys_irq_disable
```

Parameters:

None

Return Value:

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

Side Effects:

The A and X registers may be altered by this function.

l_sys_irq_restore

Description:

This function restores interrupts for the user module. It should be used in conjunction with l_sys_irq_disable. This function is essentially equivalent to the EnableInt API of most user modules. However, it should not be called when the user module is being started.

This routine is supposed to be provided by the application. However, the LINS User Module implements this routine automatically. You can modify the code in the routine if necessary.

Static C Prototype:

```
void  l_sys_irq_restore(l_irqmask previous)
```

Static Assembly:

```
mov   A, previous
lcall  l_sys_irq_restore
```

Parameters:

previous - interrupt mask that defines the digital blocks for which interrupts will be enabled.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

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 non-volatile (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 C Prototype:

```
l_u8  ld_read_configuration(l_ifc_handle iii, l_u8* const  data, l_u8* const length)
```

Dynamic Assembly:

```
mov   A, >length
push  A
mov   A, <length
push  A
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, iii
push  A
```

```
lcall  ld_read_configuration
```

Parameters:

iii - is the name of the interface handle;

data - is an array where configuration data is to be read into;

length - is a 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	This symbol is returned if the configuration data read was successful.
LD_LENGTH_TOO_SHORT	This symbol is returned if the value pointed to by the "length" pointer parameter is less than the actual length of the configuration data.

Side Effects:

The A and X registers may be altered by this function.

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 boot-up or after the master requests this to occur. Otherwise, if the slave changes its NAD and/or PID values, then master may not be able to communicate with the slave any more.

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

Dynamic C Prototype:

```
l_u8  ld_set_configuration(l_ifc_handle iii, const l_u8* const data, l_u16 length)
```

Dynamic Assembly:

```
mov   A, >length
push  A
mov   A, <length
push  A
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, iii
push  A
lcall  ld_set_configuration
;or
mov   A, [length]
push  A
mov   A, [length+1]
push  A
```

```
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, iii
push  A
lcall  ld_set_configuration
```

Parameters:

iii - is the name of the interface handle;

data - is an array of configuration data which is to be applied to the slave node;

length - is a size of configuration data in bytes.

Return Value:

The function return values are listed in the following table.

Symbolic Name	Description
LD_SET_OK	This symbol is returned if the configuration data was successfully set.
LD_LENGTH_NOT_CORRECT	This symbol is 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	This symbol is returned if the configuration data was not set correctly.

Side Effects:

The A and X registers may be altered by this function.

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.

Note that API functions that send and receive Transport Layer messages have two different formats. There is a Cooked format and a Raw format. This user module only supports using one format of the Transport Layer API functions. The API format is chosen in the Transport Layer Tab of the user module wizard.

Note To use the LIN Transport Layer API functions, Transport Layer use must be enabled on the Transport Layer Tab of the LINS User Module Wizard.

Initialization

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 is able to do any Transport Layer communication.

Dynamic C Prototype:

```
void  ld_init(l_ifc_handle iii)
```

Dynamic Assembly:

```
mov   A, iii
lcall  ld_init
```

Parameters:

iii - is the name of the interface handle.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

Raw Transport Layer API Functions

ld_put_raw

Description:

This function is used for allowing the application code to send data using the Transport layer. This function essentially just 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 multi-frame Transport Layer message requires multiple calls to this API function.

Dynamic C Prototype:

```
void  ld_put_raw(l_ifc_handle iii, const l_u8* const data)
```

Dynamic Assembly:

```
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, iii
push  A
lcall  ld_put_raw
```

Parameters:

iii - is the name of the interface handle;

data - is an array of data to be sent.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

ld_get_raw

Description:

This function is used for allowing the application code to receive data using the Transport layer. This function essentially just 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 multi-frame Transport Layer message requires multiple calls to this API function.

If the receive queue is empty no data is copied.

Dynamic C Prototype:

```
void  ld_get_raw(l_ifc_handle iii, l_u8* const  data)
```

Dynamic Assembly:

```
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, iii
push  A
lcall  ld_get_raw
```

Parameters:

iii - is the name of the interface handle;

data - is an array where the oldest received diagnostic frame data will be copied to.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

ld_raw_tx_status

Description:

The call returns the status of the last performed frame transmission on the bus when Raw type API was used.

Dynamic C Prototype:

```
l_u8  ld_raw_tx_status(l_ifc_handle iii)
```

Dynamic Assembly:

```
mov   A, iii
lcall  ld_raw_tx_status
```

Parameters:

iii - is the name of the interface handle.

Return Value:

Symbolic Name	Description
LD_QUEUE_EMPTY	The transmit queue is empty. In case previous calls to ld_put_raw, 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 can not accept further frames.
LD_TRANSMIT_ERROR	LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Side Effects:

The A and X registers may be altered by this function.

ld_raw_rx_status

Description:

The call returns the status of the last performed frame reception on the bus when Raw type API was used.

Dynamic C Prototype:

```
l_u8  ld_raw_rx_status(l_ifc_handle iii)
```

Dynamic Assembly:

```
mov   A, iii
lcall  ld_raw_rx_status
```

Parameters:

iii - is the 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:

The A and X registers may be altered by this function.

Cooked Transport Layer API Functions

ld_send_message

Description:

This function is used for allowing the application code to send data using the Transport Layer. This function 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 multi-frame Transport Layer message requires only one call to this API function. The length value must be in the range from 1 to 256 bytes.

Dynamic C Prototype:

```
void  ld_send_message(l_ifc_handle iii, l_u16 length, l_u8 NAD, const l_u8* const data)
```

Dynamic Assembly:

```
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, NAD
push  A
mov   A, >length
push  A
mov   A, <length
mov   A, iii
push  A
lcall  ld_send_message

;or
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, NAD
push  A
mov   A, [length]
push  A
mov   A, [length+1]
mov   A, iii
push  A
lcall  ld_send_message
```

Parameters:

iii - is the name of the interface handle;

length - is a size of data to be sent in bytes;

NAD - is the address of the slave node which data is sent to;

data - is an array of data to be sent.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

ld_receive_message

Description:

This function is used for allowing the application code to receive data using the Transport Layer. This function 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 multi-frame Transport Layer message requires only one call to this API function. The length value must be in the range from 1 to 256 bytes.

Dynamic C Prototype:

```
void  ld_receive_message(l_ifc_handle iii, l_u16* const length, l_u8* const NAD, l_u8*
const  data)
```

Dynamic Assembly:

```
mov   A, >data
push  A
mov   A, <data
push  A
mov   A, >NAD
push  A
mov   A, <NAD
push  A
mov   A, >length
push  A
mov   A, <length
push  A
mov   A, iii
push  A
lcall  ld_receive_message
```

Parameters:

iii - is the name of the interface handle

length - size of data to be received in bytes

NAD - is the address of the slave node which data is received from

data - array of data to be received

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

ld_tx_status

Description:

The function returns the status of the last made call to ld_send_message and the last made Transport Layer data transmission on the bus.

Dynamic C Prototype:

```
l_u8  ld_tx_status(l_ifc_handle iii)
```

Dynamic Assembly:

```
mov   A, iii
lcall  ld_tx_status
```

Parameters:

iii - is the 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 shall 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 a N_As timeout, and current message transmission will be aborted. See Section 3.2.5. of LIN 2.1 Specification.

Side Effects:

The A and X registers may be altered by this function.

ld_rx_status

Description:

The function returns the status of the last made call to ld_receive_message and the last made Transport Layer data reception on the bus.

Dynamic C Prototype:

```
l_u8  ld_rx_status(l_ifc_handle iii)
```

Dynamic Assembly:

```
mov   A, iii
lcall  ld_rx_status
```

Parameters:

iii - is the 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 a N_Cr timeout, and current message reception will be aborted. See Section 3.2.5. of LIN 2.1 Specification.
LD_WRONG_SN	The reception failed because of an unexpected sequence number.

Side Effects:

The A and X registers may be altered by this function.

Non-LIN-Specified API

LINS_bReadStatus

Description:

This function is used to return any user module status indicators that are not required or defined by the LIN specification yet are still useful.

Note: This function has the dynamic representation only.

C Prototype:

```
l_u8  LINS_bReadStatus(l_ifc_handle iii)
```

Assembly:

```
mov   A, iii
lcall  LINS_bReadStatus
```

Parameters:

iii - is the name of the interface handle;

Return Value:

This function returns one status byte. The first bit in this byte is the flag that indicates that there has been no signaling on the bus for a certain amount of elapsed time. If the elapsed time is past a certain threshold, then this flag will be set. Calling this API will clear all status bits after they are returned. The second bit is the flag that indicates that Targeted Reset service request (0xB5) was received (when J2602-1 Compliance is enabled).

7	6	5	4	3	2	1	0
0	0	0	0	0	0	Targeted Reset service request (0xB5) was received	No signal was detected on the bus for a certain amount of elapsed time

Side Effects:

The A and X registers may be altered by this function.

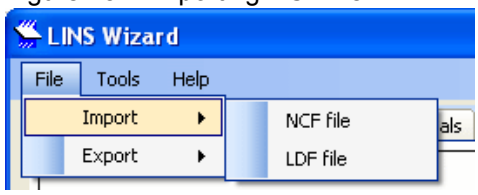
Sample Firmware Source Code

Basic Transmission Operation Example

In the following C and Assembly sample code, the LIN Slave transmits two frames: Unconditional “Frame1” and Event-Triggered “Frame2” to the LIN Master.

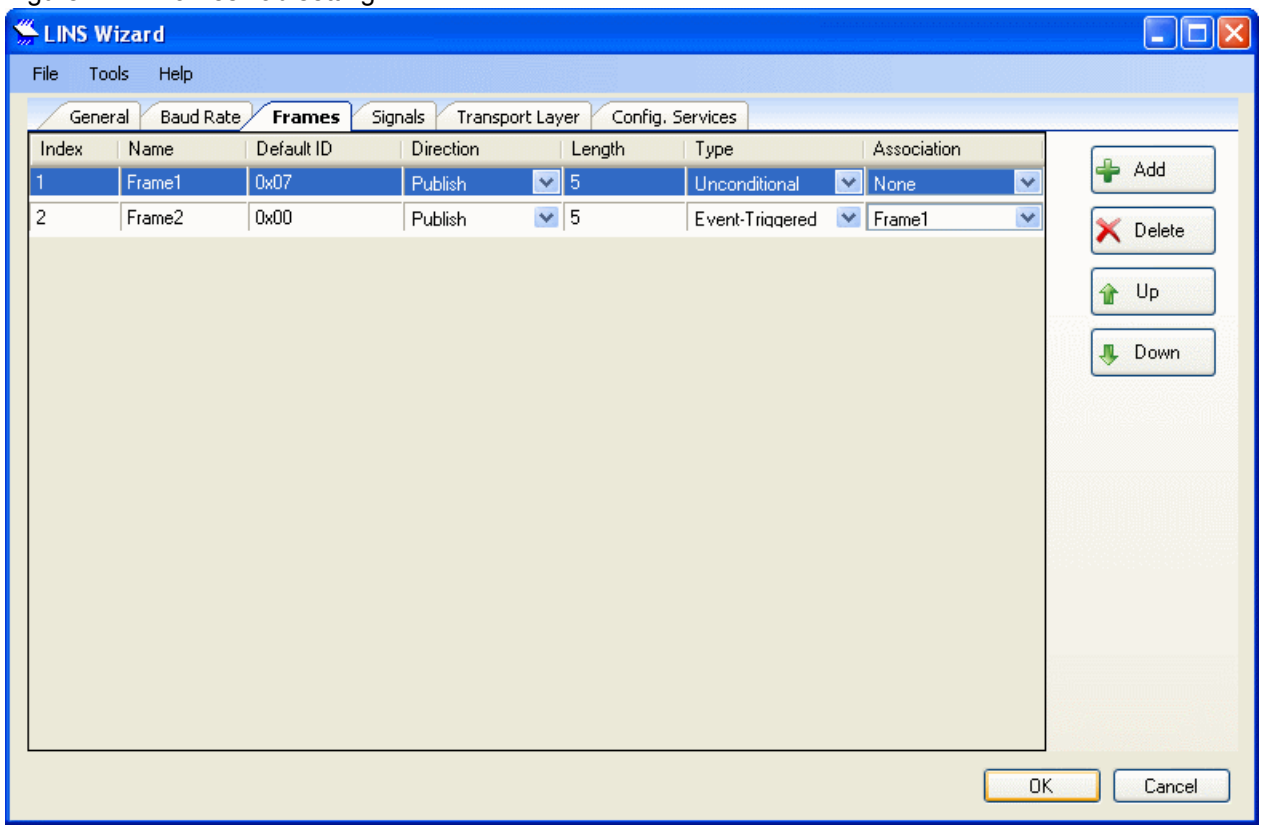
You can simply import the Node Configuration File (NCF) for Basic transmittion operation example or configure the project manually. The NCF file can be imported from the LINS User Module wizard. Go to **File > Import > NCF file** and choose the "Basic Transmission Operation example.ncf" file at: [Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples NCF. ‘Ver_X_Y’ refers to the latest version of the LINS User Module.

Figure 23. Importing NCF file



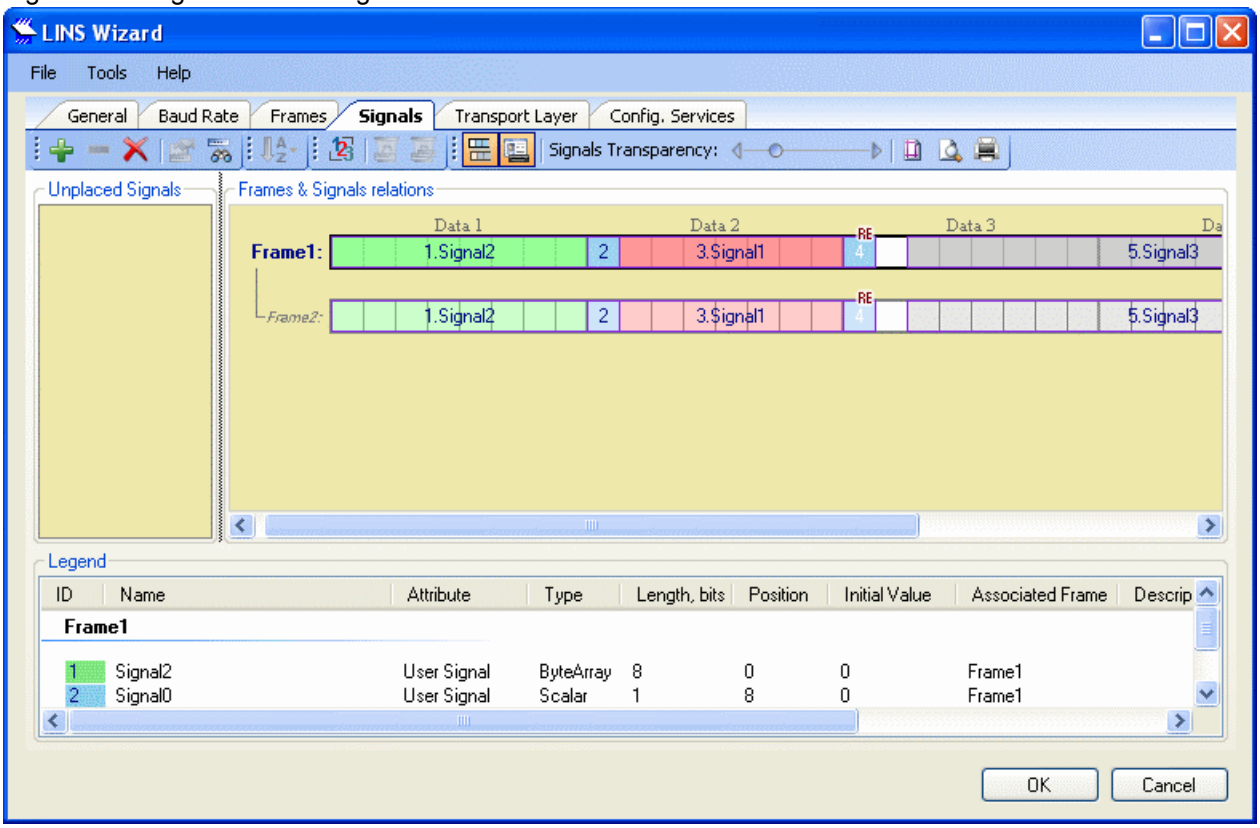
It is recommended to use NCF examples from the folder with the latest user module version number. For manual project configuration: “Frame1” and “Frame2” must be configured as shown in Figure 24.

Figure 24. Frames Tab setting



Some signals must be added to Frame1. In this example, four different signals are used. Scalar signal "Signal0" transmits 1 bit of data to the LIN master, "Signal1" is a 7-bit scalar signal, "Signal2" is a byte array signal with a length of 1 byte, and "Signal3" is a 16-bit scalar signal. The "response_error" signal is also shown. Each of these signals is shown in Figure 25.

Figure 25. Signals Tab setting



This sample code shows LINS User Module initialization. The l_sys_init and l_ifc_init_iii API functions return zero if initialization is successfully done.

The LINS User Module uses interrupt routines, that is why global interrupts should be enabled in the source code before enabling the LINS User Module.

You can operate with status word in this example and transmit any data to the LIN master.

Sample code analyzes the status of current signal flag before each transmit operation. And if LINS User Module is not in sending process now new data can be sent.

```
#include <m8c.h>         // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

l_bool fData1;          /* Bool variable for sending data to LIN master*/
l_u8   bData2;          /* Variable for sending data to LIN master*/
l_u8   abDataArray[8]; /* Variable for sending data to LIN master*/
l_u16  wLinStatus;      /* LIN status word will be saved in this variable */

void main(void)
{

    M8C_EnableGInt;          /* Enable Global interrupts */
    if(l_sys_init())         /* Call initialization function */
```

```
    {
        /* Process Core initialization error here */
    }
    if(l_ifc_init_LINS())   /* Start LINs UM */
    {
        /* Process UM initialization error here */
    }

    while(1)
    {
        wLinStatus = l_ifc_read_status_LINS();   /* Read LINs status word */

        l_bool_wr_Signal0(fData1);           /* Transmit bool data to the LIN master */

        if(l_flg_tst_Signal0())                  /* Test flag for Signal0 if its set
than we can transmit data */
        {
            l_flg_clr_Signal0();                 /* Clear Signal0 flag */
            fData1++;                            /* Prepare data for transmit */
        }

        l_u8_wr_Signal1(bData2);                /* Transmit data to the LIN master */

        if(l_flg_tst_Signal1())                  /* Test flag for Signal1 if its set
than we can transmit data */
        {
            l_flg_clr_Signal1();                 /* Clear Signal1 flag */
            bData2 = (0x55 | bData2);            /* Get some data for transmit */
        }

        l_u16_wr_Signal3(wLinStatus);          /* Sent status word to the LIN master */

        if(l_flg_tst_Signal3())                  /* Test flag for Signal3 if its set
than we can transmit data */
        {
            l_flg_clr_Signal3();                 /* Clear Signal3 flag */
        }

        l_bytes_wr_Signal2(0, 1, abDataArray);   /* Send data from data Array */

        if(l_flg_tst_Signal2())                  /* Test flag for Signal2 if its set
than we can transmit data */
        {
            l_flg_clr_Signal2();                 /* Clear Signal2 flag */
            abDataArray[0] = abDataArray[0] + 2; /* Prepare data for transmit */
        }
    }
}
```

The equivalent code written in Assembly is:

```
include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

export _main

AREA  bss (RAM,REL)

wLinStatus::     BLK 2  ;
abDataArray::    BLK 8  ;

AREA text(ROM,REL,CON)

_main:

    M8C_EnableGInt              ; Enable Global interrupts
    lcall l_sys_init             ; Call initialization function
    lcall l_ifc_init_LINS        ; Start LINs UM

.cycle:                          ; Loop label

    lcall l_flg_tst_Signal0      ; Test flag for Signal0
    cmp   A, 00h                 ; if its set than we can Transmit data
    jz    .SkipSignal0
    lcall l_flg_clr_Signal0      ; Clear Signal0 flag
    mov   A, 01h                 ; Prepare data for transmit
    lcall l_bool_wr_Signal0      ; Transmit bool data to the LIN master
.SkipSignal0:

    lcall l_flg_tst_Signal1      ; Test flag for Signal1
    cmp   A, 00h                 ; if its set than we can Transmit data
    jz    .SkipSignal1
    lcall l_flg_clr_Signal1     ; Clear Signal1 flag
    mov   A, 55h                ; Prepare data for transmit
    lcall l_u8_wr_Signal1        ; Transmit data to the LIN master
.SkipSignal1:

    lcall l_flg_tst_Signal3      ; Test flag for Signal3
    cmp   A, 00h                 ; if its set than we can Transmit data
    jz    .SkipSignal3
    lcall l_flg_clr_Signal3      ; Clear Signal3 flag
    lcall l_ifc_read_status_LINS ; Read LINs status word
    mov   X, [wLinStatus]        ; Load MSB part of status word to the X
    mov   A, [wLinStatus+1]      ; Load LSB part of status word to the A
    lcall l_u16_wr_Signal3       ; Transmit status word to the LIN master
.SkipSignal3:

    lcall l_flg_tst_Signal2      ; Test flag for Signal2
    cmp   A, 00h                 ; if its set than we can Transmit data
    jz    .SkipSignal2
    lcall l_flg_clr_Signal2      ; Clear Signal2 flag
    mov   [abDataArray], F0h
    mov   A, >abDataArray        ; Load MSB destination address. Data will be stored
to array whith this address
```

```
    push  A
    mov   A, <abDataArray        ; Load LSB destination address. Data will be stored
to array whith this address
    push  A
    mov   A, 01h                 ; Load Number of bytes to be Transmit
    push  A
    mov   A, 00h                 ; Load data offset for Transmit
    push  A
    lcall l_bytes_wr_Signal2     ; Transmit data to LIN master

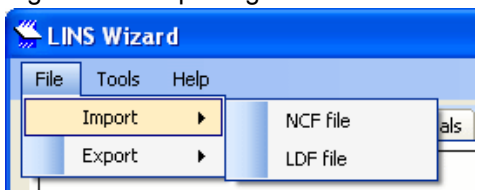
.SkipSignal2:                    ; Other user code
    jmp   .cycle
```

Basic Reception Operation Example

In the following C and Assembly sample code, the LIN Slave can receive two unconditional frames (“Frame1” and “Frame2”) from the LIN master.

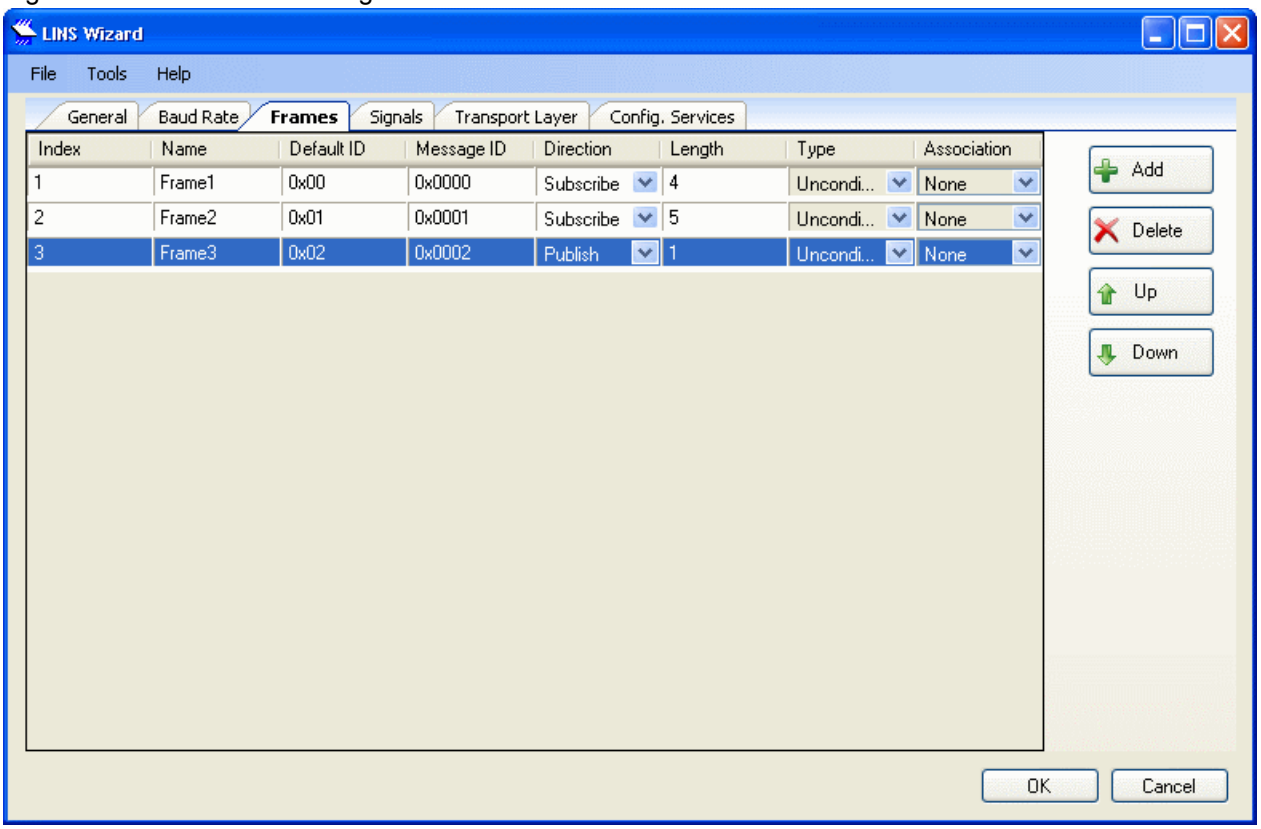
You can simply import the Node Configuration File (NCF) for Basic reception operation example or configure the project manually. The NCF file can be imported from LINS User Module wizard. Go to **File > Import > NCF file** and choose the "Basic Reception Operation example.ncf" file at: [Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples NCF. ‘Ver_X_Y’ refers to the latest version of the LINS User Module.

Figure 26. Importing NCF file



It is recommended to use NCF examples from the folder with the latest user module version number. For manual project configuration: “Frame1” and “Frame2” must be configured as shown in Figure 27.

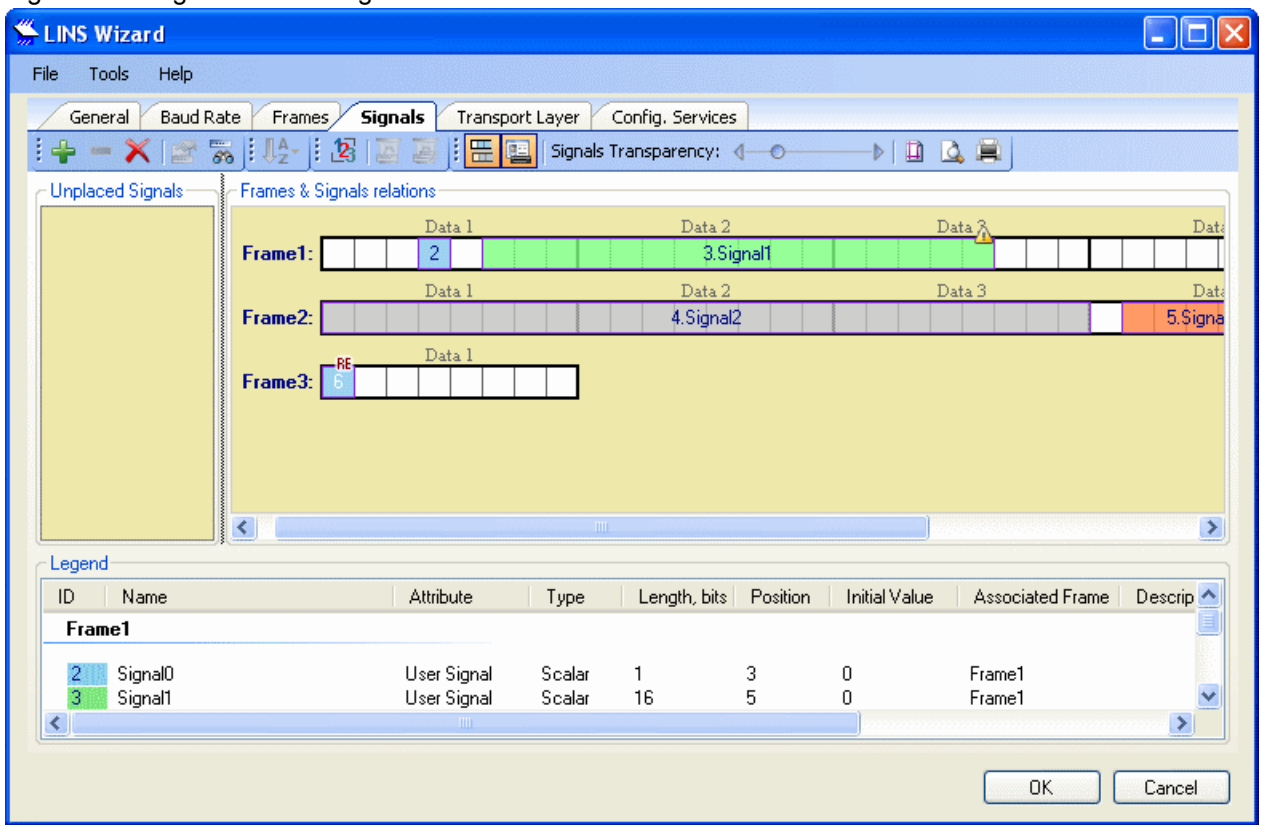
Figure 27. Frames Tab setting



Signals must be added to Frame1 and Frame2. Frame3 is used for the response_error status signal, which must be made available to the LIN master. In this example, four different "user" signals are used.

"Signal0" is a 1-bit scalar signal. "Signal1" is a 16-bit scalar signal. "Signal2" is a byte array signal with a length of 3 bytes. "Signal3" is a 5-bit scalar signal. These signals are all shown in Figure 28.

Figure 28. Signals Tab setting



This sample code shows LINS User Module initialization. The l_sys_init and l_ifc_init_iii API functions return zero if initialization is successfully done.

The LINS User Module uses interrupt routines, that is why global interrupts should be enabled in the source code before enabling the LINS User Module.

You can operate with status word in this example, and receive any data to the LIN master.

Sample code analyzes status of current signal flag (is data ready or not) before each receive operation. And if receiving is completed and data ready LINS can use new data.

```
#include <m8c.h>            // part specific constants and macros
#include "PSoCAPI.h"        // PSoC API definitions for all User Modules

l_u8  bData2;               /* Variable for collecting data recieved from LIN master*/
l_u8  abDataArray[8];       /* Variable for collecting data recieved from LIN master*/
l_u16 iData1;               /* Variable for collecting data recieved from LIN master*/
l_u16 wLinStatus;           /* LIN status word will be saved in this variable */

void main(void)
{

    M8C_EnableGInt;         /* Enable Global interrupts */
    if(l_sys_init())        /* Call initialization function */
    {
        /* Process Core initialization error here */
    }
```

```
    if(l_ifc_init_LINS())   /* Start LINs UM */
    {
        /* Process UM initialization error here */
    }

    while(1)
    {
        wLinStatus = l_ifc_read_status_LINS();      /* Read LINs status word and save
it to the variable */

       if(l_flg_tst_Signal0())                     /* Test flag for Signal0 if its set
than we can receive data */
        {
            l_flg_clr_Signal0();                    /* Clear Signal0 flag */
            if(l_bool_rd_Signal0() == 0x01)         /* Read bool data from the LIN
master and check is it true or false */
            {
                /* do some operation */
            }
        }

       if(l_flg_tst_Signal1())                     /* Test flag for Signal1 if its set
than we can receive data */
        {
            l_flg_clr_Signal1();                    /* Clear Signal1 flag */
            iData1 = l_u16_rd_Signal1();            /* Read data from the LIN master
and save tis data to the variable */
        }

       if(l_flg_tst_Signal2())                     /* Test flag for Signal2 if its set
than we can receive data */
        {
            l_flg_clr_Signal2();                    /* Clear Signal2 flag */
           l_bytes_rd_Signal2(1, 2, abDataArray);  /* Save recieved data to the user
selected data array */
        }

       if(l_flg_tst_Signal3())                     /* Test flag for Signal3 if its set
than we can receive data */
        {
           l_flg_clr_Signal3();                   /* Clear Signal3 flag */
          bData2 = l_u8_rd_Signal3();             /* Read data from the LIN master and
save this data to the variable */
        }
    }
}
```

The equivalent code written in Assembly is:

```
include "m8c.inc"       ; part specific constants and macros
include "memory.inc"    ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"   ; PSoC API definitions for all User Modules

export _main

AREA  bss (RAM,REL)

wLinStatus::      BLK 2  ;
iData1::          BLK 2  ;
bData2::          BLK 1  ; Variables which are used in the project
bDataArray::      BLK 8  ;

AREA text(ROM,REL,CON)

_main:

   M8C_EnableGInt                 ; Enable Global interrupts
   lcall l_sys_init               ; Call initialization function
   lcall l_ifc_init_LINS          ; Start LINS UM

.cycle:                           ; Loop label

   lcall l_ifc_read_status_LINS   ; Read LINs status word
   mov   [wLinStatus], X          ; Save MSB part of Satus word to the variable
   mov   [wLinStatus+1], A        ; Save LSB part of Satus word to the variable

   lcall l_flg_tst_Signal0        ; Test flag for Signal0
   cmp   A, 00h                   ; if its set than we can Recieve data
   jz    .SkipSignal0
   lcall l_flg_clr_Signal0        ; Clear Signal0 flag
   lcall l_bool_rd_Signal0        ; Read bool data from the LIN master
   cmp   A, 00h                   ; Bool data now in A. Compare is it true or not
   jz    .SkipSignal0
         ; do some operation
.SkipSignal0:

   lcall l_flg_tst_Signal1        ; Test flag for Signal1
   cmp   A, 00h                   ; if its set than we can recieve data
   jz    .SkipSignal1
   lcall l_flg_clr_Signal1        ; Clear Signal1 flag
   lcall l_u16_rd_Signal1         ; Read data from the LIN master
   mov   [iData1], X              ; Save MSB part of data to the global variable
   mov   [iData1+1], A            ; Save LSB part of data to the global variable
.SkipSignal1:

   lcall l_flg_tst_Signal3        ; Test flag for Signal3
   cmp   A, 00h                   ; if its set than we can recieve data
   jz    .SkipSignal3
   lcall l_flg_clr_Signal3        ; Clear Signal3 flag
   lcall l_u8_rd_Signal3          ; Read data from the LIN master
   mov   [bData2], A              ; Save data to the variable

.SkipSignal3:
```

```
   lcall l_flg_tst_Signal2         ; Test flag for Signal2
   cmp   A, 00h                    ; if its set than we can recieve data
   jz    .SkipSignal2
   lcall l_flg_clr_Signal2         ; Clear Signal2 flag

   mov   A, >bDataArray            ; Load MSB destination address. Data will be stor-
ed to array whith this address
   push  A
   mov   A, <bDataArray            ; Load LSB destination address. Data will be stor-
ed to array whith this address
   push  A
   mov   A, 02h                    ; Load Number of bytes to be read
   push  A
   mov   A, 01h                    ; Load data offset for reading
   push  A
   lcall l_bytes_rd_Signal2        ; Get data from LIN master

.SkipSignal2:
         ; Other user code

   jmp   .cycle
```

Transport Layer Raw API Usage Example

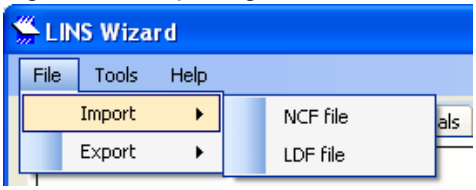
In the following sample code example the LIN Slave receives or transmits data through the LIN Transport Layer using the Raw type of the Transport Layer API functions.

You can simply import the Node Configuration File (NCF) for the Transport Layer Raw API usage example or configure the project manually.

The NCF file can be imported from the LINS User Module wizard. Go to **File > Import > NCF file** and choose the "Transport Layer Raw API example.ncf" file at:
[Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples NCF. ‘Ver_X_Y’ refers to the latest version of the LINS User Module.

Note After importing the NCF file you must enable the Transport Layer by checking the “Use Transport Layer” checkbox on the “Transport Layer” tab.

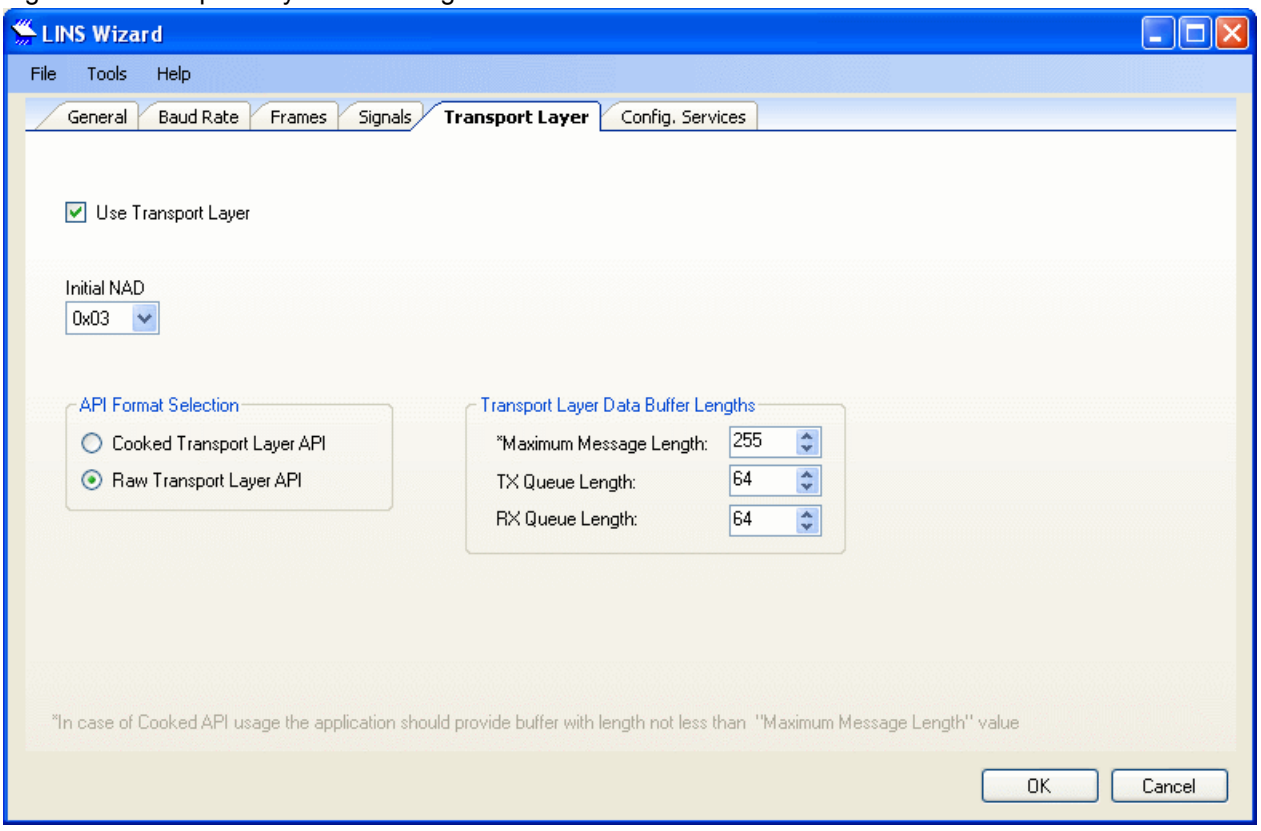
Figure 29. Importing NCF file



It is recommended to use NCF examples from the folder with the latest user module version number. For manual project configuration:

The settings of frames and signals are not important for the Transport Layer. The main Transport Layer parameters should be configured like it is shown in Figure 30.

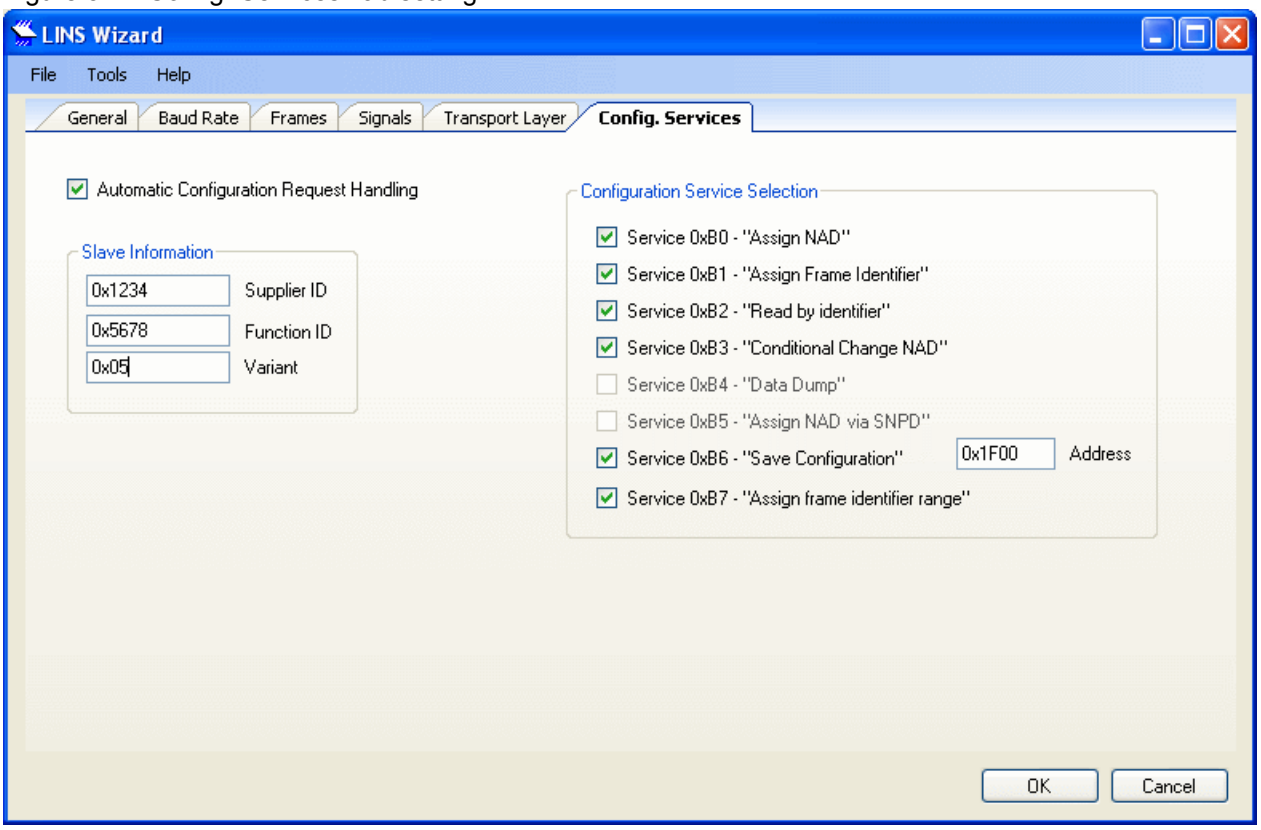
Figure 30. Trasport Layer Tab setting



You can set up the length of the TX and RX Raw buffers. The “Maximum Message Length” has no effect when Raw type of APIs is used.

If you want to use the automatic handling of Configuration services then the “Automatic Configuration Request Handling” property should be enabled on the “Config. Services” tab. You can choose any service that you want to handle automatically. Also the “Slave Information” options can be set on this tab.

Figure 31. Config. Services Tab setting



The following sample code demonstrates the use of Raw API functions:

```
#include <m8c.h>                    // part specific constants and macros
#include "PSoCAPI.h"                // PSoC API definitions for all User Modules

#define LINS_SAVE_CONF_LENGTH  6  /* Define count of data bytes which will be saved in
save configuration command */

BYTE abLength[1] = {6};       /* Length value */
BYTE abTxRxData[8];           /* Array with data */
BYTE abReadConf[8];           /* Array where LIN configuration data is stored */

void main(void)
{

    M8C_EnableGInt ;          /* Turn on global interrupts */
    if(l_sys_init())          /* Call initialization function */
    {
        /* Process Core initialization error here */
    }
    if(l_ifc_init_LINS())    /* Start LINs UM */
    {
        /* Process UM initialization error here */
    }
```

```
    ld_init(LINS_IFC_HANDLE); /* Transport layer initialization */

    while(1)
    {
        if(LINS_bReadStatus(LINS_IFC_HANDLE) & LINS_BUS_INACTIVITY_FLAG) /* Test, was
bus inactivity flag set or not */
        {
            /* Set some flag that means bus inactivity was detected */
        }

        if(l_ifc_read_status_LINS() & L_STS_SAVE_STATUS_CONF)
        {
            ld_read_configuration(LINS_IFC_HANDLE, abReadConf, abLength); /* Read LIN
configuration */
            /* Put code here to write configuration to FLASH.
             Current LIN configuration data is stored in the abReadConf array */
        }

        if(ld_raw_rx_status(LINS_IFC_HANDLE) == LD_DATA_AVAILABLE)  /* Check is data
available */
            ld_get_raw(LINS_IFC_HANDLE, abTxRxData);               /* Read data from
master */

        if(ld_raw_tx_status(LINS_IFC_HANDLE) == LD_QUEUE_EMPTY)     /* Check can we
transmitt new data to Master */
            ld_put_raw(LINS_IFC_HANDLE, abTxRxData);              /* Send message to
master with configuration data */
    }
}
```

Transport Layer Cooked API Usage Example

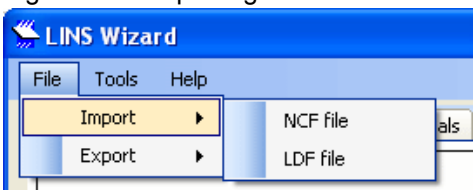
In the following sample code example the LIN Slave receives or transmits data through the LIN Transport Layer using the Cooked type of the Transport Layer API functions.

You can simply import the Node Configuration File (NCF) for the Transport Layer Cooked API usage example or configure the project manually.

The NCF file can be imported from the LINS User Module wizard. Go to **File > Import > NCF file** and choose the "Transport Layer Cooked API example.ncf" file at:
[Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples NCF\. ‘Ver_X_Y’ refers to the latest version of the LINS User Module

Note After importing the NCF file you must enable the Transport Layer by checking the “Use Transport Layer” checkbox on the “Transport Layer” tab.

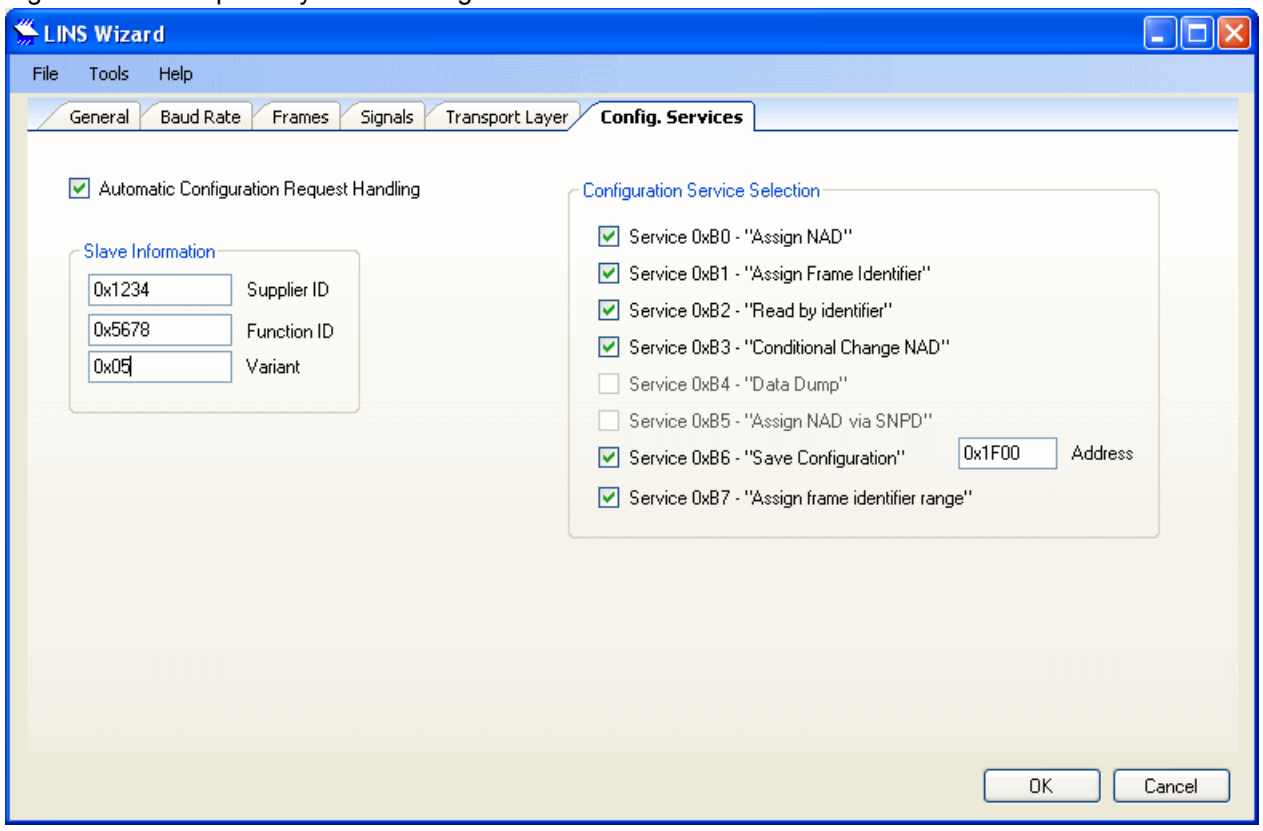
Figure 32. Importing NCF file



It is recommended to use NCF examples from the folder with the latest user module version number. For manual project configuration:

The settings of frames and signals are not important for the Transport Layer. The main Transport Layer parameters should be configured like it is shown in Figure 33.

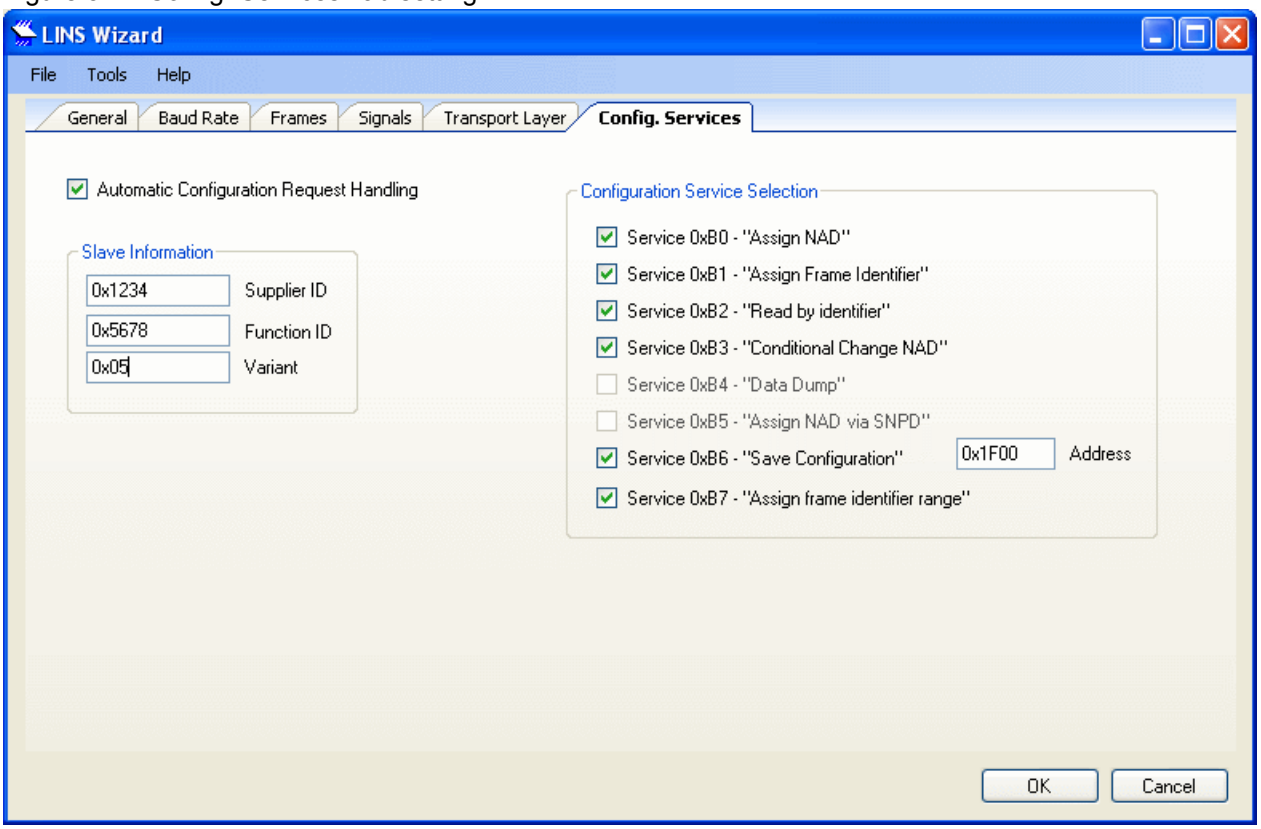
Figure 33. Transport Layer Tab setting



You can set up the “Maximum Message Length” parameter. Messages with the length larger than the “Maximum Message Length” value will be ignored by the LIN Slave.

If you want to use the automatic handling of Configuration services then the “Automatic Configuration Request Handling” property should be enabled on the “Config. Services” tab. You can choose any service that you want to handle automatically. Also the “Slave Information” options can be set on this tab.

Figure 34. Config. Services Tab setting



The following sample code demonstrates the use of Cooked API functions:

```
#include <m8c.h>                    // part specific constants and macros
#include "PSoCAPI.h"                // PSoC API definitions for all User Modules

#define LINS_SAVE_CONF_LENGTH  6  /* Define count of data bytes which will be saved in
save configuration command */

l_u8  abTxRxData[8];        /* Array with data */
l_u8  abReadConf[8];        /* Array where LIN configuration data is stored */

l_u8  abLength[1] = {6};    /* Length value */
l_u8  abNAD[1];             /* NAD value */
l_u16 awLength[1] = {6};    /* Length value */

void main(void)
{

    M8C_EnableGInt ;         /* Turn on global interrupts */
    if(l_sys_init())         /* Call initialization function */
    {
        /* Process Core initialization error here */
    }
    if(l_ifc_init_LINS())    /* Start LINS UM */
```

```
    {
        /* Process UM initialization error here */
    }
    ld_init(LINS_IFC_HANDLE); /* Transport layer initialization */

    while(1)
    {
        if(LINS_bReadStatus(LINS_IFC_HANDLE) & LINS_BUS_INACTIVITY_FLAG) /* Test, was
bus inactivity flag set or not */
        {
            /* Set some flag that means bus inactivity was detected */
        }

        if(l_ifc_read_status_LINS() & L_STS_SAVE_STATUS_CONF)
        {
            ld_read_configuration(LINS_IFC_HANDLE, abReadConf, abLength); /* Read LIN
configuration */
            /* Put code here to write configuration to FLASH.
             Current LIN configuration data is stored in the abReadConf array */
        }

        if(ld_rx_status(LINS_IFC_HANDLE) == LD_COMPLETED)                        /*
Check is LD Completed for RX */
           ld_receive_message(LINS_IFC_HANDLE, awLength, abNAD, abTxRxData);      /*
Read data from master */

        if(ld_tx_status(LINS_IFC_HANDLE) == LD_COMPLETED)                        /*
Check is LD Completed fot TX */
          ld_send_message(LINS_IFC_HANDLE, LINS_SAVE_CONF_LENGTH, 3, abTxRxData); /*
Send message to master with configuration data */
    }
}
```

NCF/LDF File Import Examples

The user module wizard settings for the example code can be automatically setup by importing example NCF or LDF files that are included with this user module.

Importing Example NCF File

1. Open LINS User Module wizard.
2. Select menu **File > Import > NCF File**.
3. In the window that opens, go to [Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples NCF. ‘Ver_X_Y’ refers to the latest version of the LINS User Module. It is recommended to use NCF examples from the folder with the latest user module version number.
4. Select "Basic Transmission Operation example" file. Click "Open" button.
5. In the popup window click "Ok" button.
6. Observe import results on the new popup window. Click "Ok" button.
7. Go to Frames Tab and observe that two frames are configured (see Figure 24).
8. Go to Signals Tab and observe that five signals are packed in "Frame1" (see Figure 25).
9. As shown in Figure 24, "Frame1" has "Default ID" 0x00 instead of 0x07. (According to LIN 2.1 specification, an NCF file does not contain information about "Default ID" values).
10. Select "Default ID" cell of frame "Frame1" and change value from 0x00 to 0x07.
11. Go to General Tab and select RxD and TxD pins.
12. Go to Baud Rate Tab and set "Nominal LIN Bus Baud Rate" value, click the "Solve" button to find necessary dividers and click "Apply" button to apply changes.
13. Click "Ok" button on the window that appears.
14. Generate project and copy "Basic Transmission Operation example" sample code into main.c.
15. Build project.

Importing Example LDF File

1. Open LINS User Module wizard.
2. Select menu **File > Import > LDF File**.
3. In opened window go to [Install path]\Common\CypressSemiDeviceEditor\Data\Stdum\LINS\Ver_X_Y\Examples LDF. ‘Ver_X_Y’ refers to the latest version of the LINS User Module. It is recommended to use LDF examples from the folder with the latest user module version number.
4. Select "Basic Transmission Operation example" file. Click "Open" button.
5. On popup window click "Ok" button.
6. Observe import results on new popup window. Click "Ok" button.
7. Go to Frames Tab and observe that two frames are configured (see Figure 24).
8. Go to Signals Tab and observe that five signals are packed in "Frame1" (see Figure 25).
9. Go to General Tab and select RxD and TxD pins.
10. Go to Baud Rate Tab and set "Nominal LIN Bus Baud Rate" value, click on "Solve" button to find necessary dividers and click "Apply" button to apply changes.
11. Click "Ok" button on the window that appears.
12. Generate project and copy “Basic Transmission Operation example” sample code into main.c.
13. Build project.

Configuration Registers

The LINS User Module uses the Timer8, TX8, and RX8 digital PSoC blocks. Each block is personalized and parameterized through a set of registers. The set of registers used by the user module with brief descriptions are given in this section. Symbolic names for these registers are defined in the user module instance's C and assembly language interface files (the ".h" and ".inc" files).

Timer8 Block Registers

- Bank 1
  - Function Register: DxCxxFN

    This register defines the settings of the Digital Basic/Communications Type 'B' Block to be the Timer8 block of the LINS User Module.
  - Input Register: DxCxxIN

    This register is used to select the data and clock inputs for the Timer8 digital block of the LINS User Module.
  - Output Register: DxCxxOU

    This register is used to control the connection of the Timer8 digital block outputs to the available row interconnect and control clock resynchronization.
- Bank 0
  - Control Register 0: DxCxxCR0

    This register is the Control Register 0 for the Timer8 digital block of the LINS User Module.
  - Counter Register: DxCxxDR0

    This register is the Data Register 0 for the Timer8 digital block of the LINS User Module.
  - Period Register: DxCxxDR1

    This register is the Data Register 1 for the Timer8 digital block of the LINS User Module.
  - Compare Register: DxCxxDR2

    This register is the Data Register 2 for the Timer8 digital block of the LINS User Module.

TX8 Block Registers

This block can operate in the Timer and TX modes. For more information refer to the Block Resources datasheet section.

- Bank 1
  - Function Register: DxCxxFN

    This register defines the settings of the Digital Basic/Communications Type 'B' Block to be the TX8 block of the LINS User Module in the Timer/TX8 mode.

- Input Register: DxCxxIN

  This register is used to select the data and clock inputs for the TX8 digital block of the LINS User Module in the Timer/TX8 mode.

- Output Register: DxCxxOU

  This register is used to control the connection of the TX8 digital block outputs to the available row interconnect and control clock resynchronization.

- Bank 0

  - Control Register 0: DxCxxCR0

    This register is the Control register for the TX8 digital block which operates in the Timer/TX8 mode.

  - Counter Register: DxCxxDR0

    This register is the Data Register 0 for the TX8 digital block which operates in the Timer/TX8 mode.

  - Period Register: DxCxxDR1

    This register is the Data Register 1 for the TX8 digital block which operates in the Timer/TX8 mode.

RX8 Block Registers

- Bank 1

  - Function Register: DxCxxFN

    This register defines the settings of the Digital Basic/Communications Type 'B' Block to be the RX8 block of the LINS User Module.

  - Input Register: DxCxxIN

    This register is used to select the data and clock inputs for the RX8 digital block of the LINS User Module.

  - Output Register: DxCxxOU

    This register is used to control the connection of the RX8 digital block outputs to the available row interconnect and control clock resynchronization.

- Bank 0

  - Control Register 0: DxCxxCR0

    This register is the Control Register 0 for the RX8 digital block of the LINS User Module.

  - Shift Register: DxCxxDR0

    This register is the Data Register 0 for the RX8 digital block of the LINS User Module.

- Buffer Register: DxCxxDR2

  This register is the Data Register 1 for the RX8 digital block of the LINS User Module.

Additional Registers

- Bank 1
  - Oscillator Control Register 0: OSC_CR0

    This register is used in case when the CPUClk Speedup parameter is enabled. For more information please see the CPUClk Speedup parameter description in the Parameters and Resources section.
  - Oscillator Control Register 1: OSC_CR1

    This register is used when the LINS User Module Wizard updates global dividers after the Baud Rate calculation and when the l_ifc_ioctl API function is called with the L_OP_SET_BAUD_RATE parameter (user changes the current Baud Rate). The LINS User Module updates this register and changes the VC1 clock divider if it is requested by the user.
  - Oscillator Control Register 3: OSC_CR3

    This register is used when the LINS User Module Wizard updates global dividers after the Baud Rate calculation and when the l_ifc_ioctl API function is called with the L_OP_SET_BAUD_RATE parameter (user changes the current Baud Rate). The LINS User Module updates this register and changes the VC3 clock divider if it is requested by the user.
  - Oscillator Control Register 4: OSC_CR4

    This register is used when the LINS User Module Wizard updates global dividers after the Baud Rate calculation and when the l_ifc_ioctl API function is called with the L_OP_SET_BAUD_RATE parameter (user changes the current Baud Rate). The LINS User Module updates this register and sets the clocking source for the VC3 clock divider.
- Bank 0/Bank 1
  - Row Digital Interconnect Logic Table Register 0 and 1: RDIxLT0, RDIxLT1

    These registers are used to select the logic function of the digital row LUTs. The LINS User Module updates these registers when the TX8 block reconfigures to the Timer8 block and vice versa. When the digital block works in the TX8 mode it should be connected to a digital row, if it works in the Timer8 mode then the digital block should be disconnected from the digital row.

Version History

Version	Originator	Description
1.0	DHA	Initial version.
1.10	DHA	1. Signal preview updated for signals with length > 3 bytes. 2. Improved firmware. 3. Added automatic Baud Rate Synchronization DRC. 4. Added Event-triggered frames to Signals tab. 5. Improved user interface. Added new features to the Signals tab in the user module wizard.

Version	Originator	Description
1.10.b	DHA	The following updates were done to this user module datasheet: a. Corrected description in "API Format Selection", "Maximum Message Length", "TX Queue Length/RX Queue Length", and "Automatic Configuration Request Handling" sections. b. Updated images.
2.00	DHA	1. Transferred the `@INSTANCE_NAME`_SLAVE_NODE_MODEL_CONFIG_DATA and `@INSTANCE_NAME`_PID_INFO_TABLE rom arrays from "AREA UserModules" to "AREA lit". 2. Added support for CY8C29466-12PVXE and CY8C29466-24PVXA parts. 3. Updated the functionality of the ld_send_message Cooked API when TX buffer is set to 256 bytes but data to send is less than 256 bytes. 4. Updated the functionality of flags for Raw TX status. 5. Changed the return value from bitfield to mutually exclusive values for the following functions: a. ld_rx_status b. ld_tx_status c. ld_raw_rx_status d. ld_raw_tx_status e. ld_set_configuration f. ld_read_configuration" 6. Updated the user module to support Japanese OS. 7. Updated the l_sys_irq_disable function to address returned value errors. 8. Corrected the calculated flash address for LINSlaveConfigData saving. 9. Fixed an issue with newly added signals selection. 10. Changed the default starting index for signals. 11. Updated the Transport Layer Usage sample code example for Cooked API functions.
3.00	DHA	1. Updated area declarations to support Imagecraft optimization. 2. Updated the Transport Layer Process MRF to address the receiving of 256 data bytes. 3. Updated the LINS User Module to address an LDF file import error. 4. Modified the wizard to export support_sid in the correct format.
3.00.b	DHA	Updated warning message for 'J2602-1 Compliance' option in user module wizard.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2010-2013 Cypress Semiconductor Corporation. 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 Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark 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.