IMPLEMENTATION

This application note presents a LIN 2.0 slave driver for an 8-bit PIC® microcontroller with on-chip Enhanced Universal Synchronous Receiver Transmitter (EUSART) peripheral. EUSART is a USART with added features like automatic detection and calibration of the baud rate, wake-up on Break reception and 13-bit Break character transmit to support LIN applications. This document also shows cases how MPLAB® Code Configurator (MCC) can be used in developing the LIN drivers. This application note focuses mainly on how to setup MCC and use the drivers generated by MCC, so the LIN system designer is able to get an application running on LIN quickly. In this application note, the PIC16F1783 is used, but any 8-bit PIC microcontroller with EUSART can be used.

The information in this application note is presented with the assumption that the reader is familiar with LIN specification v2.0. Therefore, not all details about LIN are discussed. Refer to the References section of this document for additional information.

LIN PROTOCOL

LIN (Local Interconnect Network) provides a low-cost bus communication for many networks, including automotive and appliance. LIN protocol provides system development guidelines for data communication. The key features of the LIN are:

- Low-cost, single-wire implementation
- Single master, multiple slaves – up to 16
- Slaves capable of synchronizing to any baud rate from 2K to 20K.
- Self-synchronization of slaves to master’s speed
- Data format similar to common serial UART format
- Data checksums and bit-error detection for reliable operation.

LIN 2.0 Compatibility with LIN 1.x

The LIN 2.0 specification is a superset of LIN 1.x. Nodes designed to the LIN 2.0 specification and the LIN 1.x specification will communicate with each other with a few exceptions. The LIN 2.0 master node cannot request the new LIN 2.0 features from a LIN 1.x node:

- Enhanced checksum
- Reconfiguration and diagnostics
- Automatic baud rate detection
- Response error status monitoring

A LIN 2.0 slave node will not work with a LIN 1.x master unless the slave node is reconfigured.

APPLICATIONS

The first question that should be asked is: “Will this driver work for my application?” The next few sections can help those who would like to know the answer to this question and quickly decide whether this is the appropriate driver implementation or device for their application. The important elements that have significant weight on this decision include available process time, resource usage, and bit rate performance.

Process Time

The process time is dictated predominately by bit rate, clock frequency, and code execution. This is why the EUSART module is recommended for LIN communication. This hardware resource puts more processing in hardware and less demand for firmware. Thus, the average Firmware Process Time is relatively low. Table 1 shows the approximate average Firmware Process Time for Fosc equal to 16 MHz.

<table>
<thead>
<tr>
<th>Bit Rate (bps)</th>
<th>Firmware Process Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2.7</td>
</tr>
<tr>
<td>2400</td>
<td>3.2</td>
</tr>
<tr>
<td>9600</td>
<td>13</td>
</tr>
<tr>
<td>10417</td>
<td>14</td>
</tr>
<tr>
<td>20000</td>
<td>27</td>
</tr>
</tbody>
</table>
When the LIN bus is IDLE, the driver uses even less process time, approximately 1% at 16 MHz.

**Resource Usage**

The resource required on the PIC16F1783 (or any other devices which have EUSART module) are:

- EUSART – used for communication
- Timer1 – used for bus and frame timing
- Program Memory – 885 Words for Firmware LIN Drivers
- Data Memory – 22 Bytes

**Bit Rate**

The driver is designed to achieve the maximum bit rate defined by the LIN specification: 20000 bps. To ensure proper detection of Break and Sync, the internal clock needs to be accurate and synchronized. The clock frequency is defined by the master. All slaves synchronize their internal clock during the sync field inside the header. For the complete LIN-frame (header plus response) the allowed clock deviation is < ±2%, according to the LIN specification.

**Summary**

The LIN Slave driver takes advantage of the EUSART module to handle most of the otherwise demanding processing, so process time is of little concern. Timer1 is the only other resource, and it interrupts at the bit rate. Therefore, the driver can run virtually transparent in the background without significant interference to the application.

Since most of the resources, including process time, are available, this driver is well suited for high demand, high-process time applications.

**SETTING UP THE DRIVER**

To set up the firmware and start building an application, the user can configure and generate the module drivers for EUSART and Timer1 using the MLPAB Code Configurator. The user needs to include the LIN driver files, provided in the appendix, into their project.

Prior to setting up the driver, the MPLAB X IDE, XC8 compiler, and MPLAB Code Configurator (MCC) plug-in must be installed. Here are the basic steps required to set up your project:

1. Create a new MPLAB X IDE project or open an existing project.
2. Open the MCC plug-in tool. For this, in the menu bar of MPLAB X IDE, go to **Tools-> Embedded** and click **MPLAB Code Configurator**.
3. Clicking the **System** module in the Project Resources area invokes the Configuration bits setting GUI in the Composer area where device Configuration bits can be set. Also enter the system frequency if EC/HS/XT oscillator is used. Refer to Figure 1 and look at the encircled items.
4. From the Device Resources area, select Timer1 and EUSART asynchronous (NRZ) modules. When double clicking on these modules, the modules appear on the Project Resources area. Any other modules that are used in the application can also be moved into the Project Resource area.
5. In the Project Resource area, select EUSART asynchronous (NRZ) module and configure it as shown in Figure 2. Some microcontrollers (e.g., PIC16F1708) include a feature called Peripheral Pin Select (PPS) that allows the programmer to assign many of the peripherals offered by the microcontroller to any pins. The programmer has to lock the GPIO pins in the pin manager window to configure the GPIO pins as EUSART TX and RX pins.
6. In the Project Resource area, select Timer1 module and configure it as shown in Figure 3. The timer period value is equal to the bit rate value required for LIN communication. The following equation gives the bit time based on EUSART baud rate for Timer1.

\[
T = 0xFFFF - \frac{(BRG+1)}{PR}
\]

The value \(T\) represents the 16-bit value loaded in the TMR1 register. The value BRG represents the 16-bit value of the baud rate timer (SPBRGH: SPBRGL). The value PR represents the Timer1 prescaler value.
FIGURE 1: SYSTEM MODULE GUI

FIGURE 2: EUSART ASYNCHRONOUS (NRZ) CONFIGURATION GUI
7. Also configure the other modules that are moved into the Project Resource area.

8. In the Project Resource area, select the Interrupt Manager module and change the interrupt priority as per the application requirements, as can be seen in Figure 4.

9. When all the modules configurations are done, click on the Generate Code button at the top of the Composer area to generate the drives and initializers. Generated drivers will be included in the active MPLAB X IDE project (see Figure 5).

10. Add the LIN driver files to your project: lin.c, lin.h, lin_app.c, and lin_app.h. These files are available in an example project.
11. The `mcc.h` and `mcc.c` files include the definitions of Configuration bits and the oscillator_initializer. It also includes the system_initializer function which needs to be called in the application program that internally calls other required initializers like EUSART initializer and Timer1 initializer.

12. The `pin_manager.h` and `pin_manager.c` files include the pin manager initializer functions based on the configurations which were made in the pin manager GUI.

13. The `interrupt_manager.h` and `interrupt_manager.c` files include interrupt initializer and Interrupt Service Routines.

14. The `tmr1.h` and `tmr1.c` files include Timer1 configuration functions.

15. The `eusart.h` and `eusart.c` files include EUSART configuration functions.

16. The `lin.h` and `lin.c` files include LIN configuration functions. The `lin.h` contains the definition of new types and the prototypes of the functions defined in `lin.c`.

17. The `lin_app.h` and `lin_app.c` files are the application interface and should be filled by the user. Edit these files to respond to the appropriate IDs. This could be a table or some simple compare logic. In this way, the user can define the LIN communication of his application.
18. Add any additional application related files. The example uses lin_xxxevent.c for application related functions related to specific IDs.
19. The main.c file contains the entry point in program. Edit this file as required for the application. Make sure that the interrupt is setup correctly, and initialize the driver. Also, ensure any external symbols are included. For reference, a complete application provided in the appendix, is built to with the LIN driver.

The Project

The first step is to setup the project in MPLAB X IDE. Figure 5 shows an example of how the project setup looks like. The following files are required for the LIN driver to operate:
- lin.c – the driver file
- lin.h – the header file for the driver configuration
- lin_app.h – the node configuration file
- lin_app.c – the LIN event handling file

Any additional files are defined by the system designer for the specific application. For example, Figure 5 lists these project files as lin_xxxevent.c, where xxx represents the LIN ID event name. This is simply a programming style that separates ID handling into individual objects, thus making the project format easier to understand.

The Main Object

The main.c module contains the entry point into the program, which is where the driver, hardware, and variables should be initialized. To initialize the LIN driver, call the LIN_Initializer function.

Within the main function, the driver function, LIN_Driver, should be called as shown in Example 1. Within the function, read the receive buffer and process the data.

EXAMPLE 1: MAIN ENTRY POINT

```c
void main(void)
{
    //System Initialization

    //LIN Driver Initialization
    LIN_Initializer();

    while(1)
    {
        //Check Data Received
        if(EUSART_DataReady)
        {
            LIN_Driver();
        }
        //Add application code here
    }
}
```

Modify the EUSART_Receive_ISR () function which is in eusart.c, as shown in Example 2 to enable auto baud feature.

EXAMPLE 2: EUSART RX ISR

```c
void EUSART_Receive_ISR(void)
{
    //overrun check code
    // LIN Implementation

    if(!EUSART_HasBreakDetected())
    {
        while(!EUSART_HasBreakDetected());
        BAUD1CONbits.ABDEN = 1;
    }
    //read buffer code
}
```

In eusart.h, add the following macros to enable the auto wake-up feature and read the STATUS register.

```c
#define EUSART_Status( )          (RC1STA)
#define EUSART_EnableAutoWakeUp( )
                      (BAUD1CONbits.WUE = 1)
#define EUSART_HasBreakDetected( )
                      (! (BAUD1CON & 0x02))
```

The timekeeper function is also placed in the 'TM1_ISR ()' function, which is in the tmr1.c file. The example firmware uses Timer1 for bus time out; however, the LIN designer can choose any timer. Example 3 shows the placement within the interrupt.

EXAMPLE 3: TMR1 ISR

```c
void TM1_ISR(void)
{
    //Clear the TMR1 interrupt flag
    PIR1bits.TMR1IF = 0;

    TMR1 += timer1ReloadVal;

    //Add your TMR1 interrupt custom code
    LIN_UpdateTimer();
}
```
Definitions

There are a few compile time definitions, all of them located in `lin.h`, that are used to setup the system. Table 2 lists and describes these definitions.

<table>
<thead>
<tr>
<th>Definition Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_IDLE_TIME</td>
<td>25000</td>
<td>This value is the maximum IDLE bus time. The LIN specification defines this to be 25000.</td>
</tr>
<tr>
<td>MAX_HEADER_TIME</td>
<td>39</td>
<td>This value is the maximum allowable header time. The specification defines this to be 49; however, timing does not start until after the first byte (break), so it is actually 39 (10 less than the definition).</td>
</tr>
<tr>
<td>MAX_TIME_OUT</td>
<td>128</td>
<td>This is the maximum time allowed to wait after the wake-up request has been made.</td>
</tr>
</tbody>
</table>

LIN Events

LIN event functions are where the ID is decoded to determine the next action: what to transmit, what is the device expected to receive, and how many bytes to transmit/receive. The designer should edit or modify the event function to handle specific LIN IDs. One possibility is to set up an ID table, which is useful for applications that require responding to multiple IDs. Another option is to setup some simple compare logic. The example code provided in appendix is based on compare logic.

ID Modules

The example firmware uses separate ID files for individual handling of IDs and their associated functions. The most important part to remember is to include all of the external symbols that are used. The symbols used by the driver are in `lin_app.h`, which should be included in every application module.

The function in `lin_app.c` is used to setup the driver to handle the ID event. The user can add this function to handle additional ID event.

USING THE DRIVER

After setting up a project with the LIN driver’s necessary files, it is time to start using the driver. This section presents information about using the driver. The important information addressed is:

- Handling communication flags
- Handling error flags
- State flags within the driver
- LIN ID events
- Bus wake-up

The source code provided is a simple example on using the LIN driver in an application.

Communication Flags

There are two flags that indicate when the driver has successfully completed transmission or reception of data. The complete flag is set when data has been transmitted or received without error. This flag must be cleared by the user after it is handled. Likewise, the pending flag indicates the transmit or receive operation is going on. The pending flag must also be cleared by the user after it is handled.

Error Flags

Certain error flags are set when expected conditions are not met. For example, if the slave failed to match received PID with calculated PID, a parity error flag will get set in the driver. Errors are considered fatal until they are handled and cleared. Thus, if the error is never cleared, then the driver will ignore incoming data.

The following code, shown in Example 4, demonstrates how to handle errors within the main program loop. This example only shows a response to a bus time-out error. This same concept can be applied to other types of errors.

EXAMPLE 4: ERROR HANDLING

```c
void main(void)
{
    //System Initialization

    //LIN Driver Initialization
    LIN_Initializer( );
    while(1)
    {
        //Check Data Received
        if(EUSART_DataReady)
        {
            LIN_Driver( );
        }
        //Check Bus Timeout error
        if(l_errflgs_b.bits.LE_TIMEOUT==1)
        {
            LIN_GotoSleep();
        }
    }
}
Driver States

The LIN driver uses different states to remember where it is between received bytes. After a byte is received, the driver uses these states to decide what the next unexecuted state is, then change to that state.

ID Events and Functions

For each ID there exists an event function. The event function is required to tell the driver how to respond to the data following the ID. For example, the driver needs to prepare to receive or transmit data. Also, how much data is expected to be received or transmitted.

For successful operation, three variables must be initialized:

- A pointer to data memory
- Frame time
- Data length

Waking the Bus

A LIN bus wake-up function, LIN_SendWakeupSignal, is provided for applications that need the ability to wake the bus up. Calling this function will broadcast the wake-up request character.

IMPLEMENTATION

There are four functions found in the associated example firmware that control the operation of the LIN interface:

- LIN Transmit/Receive Driver
- LIN Timekeeper
- LIN Initialization
- LIN Wake-up

The Driver

The Enhanced USART module is the key element used for LIN communications. Using the EUSART module as the serial engine for LIN has certain advantages. One particular advantage is that it puts serial control in the hardware, rather than in the software. Thus, miscellaneous processing can be performed while data is being transmitted or received. According to this, the Slave Node LIN Protocol Driver is designed to run in the background, basically as a daemon.

The driver is interrupt driven via the EUSART receive interrupt. Because of the physical feedback nature of the LIN bus (Figure 6), a EUSART receive interrupt will occur regardless of transmit or receive operation. In addition, errors flags are maintained to indicate errors during transmit or receive operations.
STATES AND STATE FLAGS

The LIN driver uses state flags to indicate current state of the LIN transceiver. When an interrupt occurs, the driver uses these flags to decide what the next unexecuted state is, and then jumps to that state. Figure 7 and Figure 8 outline the program flow through the different states. Table 4 lists and describes the states definitions.

FIGURE 7: RECEIVER HEADER PROGRAM FLOW

[Diagram of program flow through different states, including decisions for Interrupt, Requesting Wake-up, Update Bus Timer, Have Break, Have Sync, Have ID, Test ID, Determine RX or TX, Measure and Test, Set Flags, and flow to LIN Message Flow Chart and Finish.]
FIGURE 8: TRANSMIT/RECEIVE MESSAGE PROGRAM FLOW

A (From LIN Header Flow Chart)

TX or RX?

RX

Got Whole Message?

No

Test, Set Flags

Yes

Read Checksum

TX

Read Back?

No

Test, Set Flags

Yes

Sent Whole Message?

No

Test, Set Flags

Yes

Sent Checksum?

No

Test, Set Flags

Yes

Read State Flags

Finish
Time-out Implementation

The time-out feature is user-selectable and dependent on the baud rate chosen by the user. If enabled, Timer1 is used to track the calculated maximum response time allowed. The LIN specification 2.0 describes the frame slot time allocation requirements as follows:

- The nominal value for transmission of a frame matches the number of bits transmitted excluding any response space, byte spaces or inter-frame space.

\[
T_{\text{Header\_Nominal}} = 34 \times T_{\text{Bit}}
\]

\[
T_{\text{Response\_Nominal}} = 10 \times (N_{\text{Data}} + 1) \times T_{\text{Bit}}
\]

\[
T_{\text{Frame\_Nominal}} = T_{\text{Header\_Nominal}} + T_{\text{Response\_Nominal}}
\]

The maximum value between the bytes is an additional 40% of time allowance compared to the nominal value.

\[
T_{\text{Header\_Maximum}} = 1.4 \times T_{\text{Header\_Nominal}}
\]

\[
T_{\text{Response\_Maximum}} = 1.4 \times T_{\text{Response\_Nominal}}
\]

\[
T_{\text{Frame\_Maximum}} = T_{\text{Header\_Maximum}} + T_{\text{Response\_Maximum}}
\]

- TBit is the time required to transmit one bit
- NData is the number of data bytes in the frame

When using time out, the timing starts with the max header time calculation that begins after receiving the first 10 TBits of the Sync Break. After the Sync Break is verified by checking for 0x00 in the receiver buffer and the frame error flag bit, Timer1 is enabled and written with the value of \((T_{\text{Header\_Maximum}} - 10 \times T_{\text{Bit}})\). The header time-out count ends after the identifier has been received or a time-out event. The identifier provides information about the length of the response. Therefore response time-out value can be calculated by first multiplying the total number of bytes (data plus one byte of checksum) and TBit then add the remaining time left that was not used from the header time out. Figure 9 shows the flow of timekeeping function.

FIGURE 9: TIMEKEEPING PROGRAM FLOW
LIN Slave Driver File Description

lin.h

The lin.h contains the definition of new types and the prototypes of the functions defined in lin.c.

TABLE 3: VARIABLES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readback_b</td>
<td>1</td>
<td>Holding the transmitted data to be compared with received data for bit error detection</td>
</tr>
<tr>
<td>checksum_w</td>
<td>2</td>
<td>Used by the driver to calculate checksum for transmit and receive</td>
</tr>
<tr>
<td>datalength_b</td>
<td>1</td>
<td>Used by the driver to maintain a message data count</td>
</tr>
<tr>
<td>handle_b</td>
<td>1</td>
<td>Holding register for the received identifier byte</td>
</tr>
<tr>
<td>bufferptr_b</td>
<td>1</td>
<td>Pointer to a storage area used by the driver. Data is either loaded into or read from memory depending on the identifier</td>
</tr>
<tr>
<td>l_state_b</td>
<td>1</td>
<td>Indicate what state the LIN bus is in</td>
</tr>
<tr>
<td>l_errflgs_b</td>
<td>1</td>
<td>Contains the error information about the LIN bus</td>
</tr>
<tr>
<td>l_comflgs_b</td>
<td>1</td>
<td>Indicate the current bus activity</td>
</tr>
</tbody>
</table>

TABLE 4: LIN STATES

<table>
<thead>
<tr>
<th>Enum. Value</th>
<th>State Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIN_ST_SYNCHBREAK</td>
<td>In this state slave ready to receive break field</td>
</tr>
<tr>
<td>1</td>
<td>LIN_ST_SYNCHBYTE</td>
<td>In this state slave receives the sync byte</td>
</tr>
<tr>
<td>2</td>
<td>LIN_ST_IDENTIFIER</td>
<td>In this state slave receives the protected ID</td>
</tr>
<tr>
<td>3</td>
<td>LIN_ST_XMIT</td>
<td>In this state slave responds to the master request</td>
</tr>
<tr>
<td>4</td>
<td>LIN_ST_RECEIVE</td>
<td>In this state slave receives the data from master</td>
</tr>
<tr>
<td>5</td>
<td>LIN_ST_SLEEP</td>
<td>In this state slave ready to receive wake-up signal</td>
</tr>
</tbody>
</table>
### REGISTER 1: ERROR FLAGS

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>State 1</th>
<th>State 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>LE_TIMEOUT: Time-Out Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 7 = Bus activity time-out error occurred</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = No bus activity time-out error occurred</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LE_NORESPONSE: No Response Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 6 = No response from the master during a Receive mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Response from the master during a Receive mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LE_INCOMPLETE: Incomplete Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 5 = Incomplete transmission or reception of frame occurred</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Complete transmission or reception of frame occurred</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LE_BITERR: Bit Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 4 = Bit error occurred during communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = No bit error occurred during communication</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LE_CHKSUMERR: Checksum Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 3 = Checksum error occurred during a Receive mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = No checksum error occurred during a Receive mode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LE_PARITYERR: Parity Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 2 = Parity error occurred in received PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = No parity error occurred in received PID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LE_SYNCERR: Synchronization Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 1 = Sync field is not detected</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Sync field is detected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>LE_BRKERR: Break Error bit</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>bit 0 = Break field is not detected</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Break field is detected</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- **R** = Readable bit
- **W** = Writable bit
- **U** = Unimplemented bit, read as ‘0’
- ‘-n’ = Value at POR
- ‘1’ = Bit is set
- ‘0’ = Bit is cleared
- **x** = Bit is unknown

---

© 2002-2015 Microchip Technology Inc.
REGISTER 2: STATUS FLAGS

<table>
<thead>
<tr>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>R-W/0-0</th>
<th>R-W/0-0</th>
<th>R-W/0-0</th>
<th>R-W/0-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Goto_sleep</td>
<td>Overrun</td>
<td>Successful_transfer</td>
<td>Error_In_Response</td>
</tr>
</tbody>
</table>

Legend:
R = Readable bit  W = Writable bit  U = Unimplemented bit, read as '0'
-n = Value at POR  '1' = Bit is set  '0' = Bit is cleared  x = Bit is unknown

bit 15-8    LastFrame_PID<7:0>: Last Frame Protected Identity
Last frame protected identity is the protected identity last detected on the bus and processed in the node. If overrun is
set one or more values of last frame protected identity are lost; only the latest vale is maintained.

bit 7-4    Reserved

bit 3    Goto_Sleep: Go to Sleep Flag bit
1 = A GOTO Sleep command has been received since the previous call to read status function
0 = A GOTO Sleep command has not been received

bit 2    Overrun: Overrun Flag bit
1 = Two or more frames are processed since the last call to read status function
0 = No overrun occurred

bit 1    Successful_transfer: Successful Transfer Flag bit
1 = One (or multiple) frame responses has been processed without an error since the last call to read status function
0 = Frame transfer is not completed

bit 0    Error_In_Response: Error in Response Flag bit
1 = One (or multiple) frames processed by the node had an error in the frame response section (e.g., checksum error,
framing error, etc.); since the previous call to read status function
0 = No frame response error
lin.c

The lin.c file has all the functions required for the operation of the LIN slave. This section describes each API function and includes comments about how to use the function.

- **LIN_Initializer**
  - **C Prototype:** void LIN_Initializer (void)
  - **Description:** Initializes all the required parameters of LIN slave. Call this function in the main function after the system initialization to start the LIN operation.
  - **Parameters:** None
  - **Returns:** None

- **LIN_Driver**
  - **C Prototype:** void LIN_Driver (void);
  - **Description:** This function receives the LIN signals and changes the state according to the current LIN signals.
  - **Parameters:** None
  - **Returns:** None

- **LIN_ReadStatus**
  - **C Prototype:** lin_status LIN_ReadStatus(void);
  - **Description:** The call returns a 16-bit status word.
  - **Parameters:** None.
  - **Returns:** 16-bit status word

- **LIN_SendWakeupSignal**
  - **C Prototype:** void LIN_SendWakeupSignal (void);
  - **Description:** Generates a wake-up command on the bus. This function sends a wake-up character on the LIN bus, which will simulate a wake-up call.
  - **Parameters:** None
  - **Returns:** None
• **LIN_AssignFrameID**
  
  **C Prototype:** `uint8_t LIN_AssignFrameID (uint8_t *l_NAD)`
  
  **Description:** This function assigns the protected identifier of a frame in the slave node with the address NAD and the specified supplier ID. The frame changed shall have specified message ID and will after the call have PID as the protected identifier.
  
  **Parameters:** None
  
  **Returns:** 8-bit Value

• **LIN_AssignNAD**
  
  **C Prototype:** `uint8_t LIN_AssignNAD (uint8_t *l_NAD)`
  
  **Description:** This function assigns the NAD (node diagnostic address) of all slave nodes that matches the NAD, the supplier identity code and the function identity code. The new NAD of those nodes will be new NAD.
  
  **Parameters:** None
  
  **Returns:** 8-bit Value

• **LIN_ReadByID**
  
  **C Prototype:** `uint8_t LIN_ReadByID (uint8_t *l_NAD)`
  
  **Description:** This function requests the node selected with the NAD to return the property associated with the ID parameter. Refer to Table 5.
  
  **Parameters:** None
  
  **Returns:** 8-bit Value

### TABLE 5: POSSIBLE VALUES FOR ID

<table>
<thead>
<tr>
<th>ID</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LIN Product Identification</td>
</tr>
<tr>
<td>1</td>
<td>Serial Number</td>
</tr>
<tr>
<td>2 – 15</td>
<td>Reserved</td>
</tr>
<tr>
<td>16 – 31</td>
<td>Message ID 1…16</td>
</tr>
<tr>
<td>32 – 63</td>
<td>User defined</td>
</tr>
<tr>
<td>64 – 255</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

- **Parameters:** None
- **Returns:** 8-bit Value

**lin_app.h**

This file is an extract of the LIN Description File (LDF) that represents the node configuration.

The file is structured as follows:

- Frame Definition
- Frame Union Definition
- Reserved buffer size for data exchange

**Frame Definition**

- `#define <Frame_Name>_Index value`
  
  Defines the Index value. Value is a user-defined incremental integer.
  
- `#define <Frame_Name>_ID value`
  
  Defines the frame identifier; value is in the range of 0 to 63(0x3F)
  
- `#define <Frame_Name>_PID value`
  
  Defines the protected frame identifier. See Appendix 7.2, located in "LIN Specification Package Revision 2.0" [1].
  
- `#define <Frame_Name>_Len value`
  
  Defines the response frame length without the checksum.

- `#define <Frame_Name>_MsgID value`
  
  Defines Message identifier for the each event of the slave node.

**lin_app.h**

This file is the application interface and should be filled by the user. In this way, the user can define the LIN communication of his application.

The `lin_app.c` file consists of:

**The ID Table variable**

You fill this variable to define the identifiers of the LIN frames that the application has to handle.
Each member of this list corresponds to a LIN frame and its corresponding identifier. Each member is of type `t_id_list` and has to be defined in the following way:

**EXAMPLE 5: ID LIST**

```c
typedef struct
{
    uint8_t *bufferptr;
    uint8_t len;
    uint8_t *idptr;
    t_msg_direction direction;
    uint16_t MsgID;
}t_id_list;
```

- **bufferptr** represents the address of the memory allocated to frame.
- **len** represents the number of data bytes of the corresponding frame. It can be set between 0 to 8.
- **idptr** represents the address of the variable to store the frame identifier.
- **direction** represents the data flow direction, is of type `t_message_direction` and should therefore be set to `ID_REQUEST` and `ID_RESPONSE` for data being request by the master and to sent by the slave.
- **MsgID** represents the message identifier of the each frame.

**REFERENCES**


**APPENDIX A: SOURCE CODE**

Due to size considerations, the complete source code for this application note is not included in the text. A complete version of the source code, with all required support files, is available for download as a Zip archive from the Microchip web site at: [www.microchip.com](http://www.microchip.com).
Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks
The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, flexPWR, JukeBlox, KEELOG, KEELOG logo, Kleer, LANCheck, MediaLB, MOST, MOST logo, MPLAB, OptoLyzer, PIC, PICSTART, PIC<sup>®</sup> logo, RightTouch, SpyNIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

The Embedded Control Solutions Company and mTouch are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, ECAN, In-Circuit Serial Programming, ICSP, Inter-Chip Connectivity, KleerNet, KleerNet logo, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, RightTouch logo, REAL ICE, SFI, Serial Quad I/O, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.

GestIC is a registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

All other trademarks mentioned herein are property of their respective companies.

© 2002-2015, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.


QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949 ==