The LIN protocol was originally designed by a group of European carmakers to be used as a low-cost, short distance, low-speed network for automotive applications (see Appendix C: “References”).

The main characteristics of the LIN protocol are:
- Serial communication
- Single master, multiple slave concept
- Low-cost, one-wire implementation
- Speed up to 20 Kbit/s
- Self-synchronization (on the slave side)
- Ensured latency time in transmission

This application note presents a Microchip Application Maestro™ compatible interrupt driven implementation of the Master Side Driver of the LIN protocol in a PIC18F device in C language (Microchip and HI-TECH ‘C’ compatible), which takes advantage of the new features provided by the PIC18 Enhanced USART module.

FILES

The implementation presented in this application note is based on the LIN Specification Package Version 1.3. This specification adheres to Microchip's Application Maestro standard and contains the following files:
- `ELINMInt.c` – C source file, contains all functions and variables used by the LIN protocol.
- `ELINMInt.h` – Header file, contains constants, unions and structures definitions, function prototypes and macro definitions used by the LIN protocol.
- `ELINMInt.def` – Contains the definitions used to configure the LIN protocol.
- `ELINMInt.ex.txt` – Example of code using the driver.

Author: Caio Gübel
Microchip Technology Inc.
**MACROS**

The following macros are defined in the ELINMInt.h file:

<table>
<thead>
<tr>
<th>Name</th>
<th>Usage</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mELINMIntInitialize</td>
<td>Initialization</td>
<td>Initializes EUSART and driver’s variables</td>
<td>3</td>
</tr>
<tr>
<td>mELINMIntTXBufferAvailable</td>
<td>Transmission</td>
<td>Checks for available transmission buffers</td>
<td>4</td>
</tr>
<tr>
<td>mELINMIntGetTXPointer</td>
<td>Transmission</td>
<td>Sets a tag and returns a buffer data pointer</td>
<td>5</td>
</tr>
<tr>
<td>mELINMIntSendMessage</td>
<td>Transmission</td>
<td>Requests the transmission of a message</td>
<td>6</td>
</tr>
<tr>
<td>mELINMIntTXStatus</td>
<td>Transmission</td>
<td>Returns the status of a sent message</td>
<td>8</td>
</tr>
<tr>
<td>mELINMIntMessageSent</td>
<td>Transmission</td>
<td>Checks for end of a message transmission</td>
<td>9</td>
</tr>
<tr>
<td>mELINMIntTXErrorDetected</td>
<td>Transmission</td>
<td>Checks for transmission errors</td>
<td>10</td>
</tr>
<tr>
<td>mELINMIntTXErrorTag</td>
<td>Transmission</td>
<td>Returns the tag of a message with error</td>
<td>11</td>
</tr>
<tr>
<td>mELINMIntTXErrorCode</td>
<td>Transmission</td>
<td>Returns the error code of a transmission</td>
<td>12</td>
</tr>
<tr>
<td>mELINMIntRXBufferAvailable</td>
<td>Reception</td>
<td>Checks for available reception buffers</td>
<td>13</td>
</tr>
<tr>
<td>mELINMIntReceiveMessage(tag,i,s)</td>
<td>Reception</td>
<td>Requests a slave to send a message</td>
<td>14</td>
</tr>
<tr>
<td>mELINMIntMessageReceived</td>
<td>Reception</td>
<td>Checks for end of reception process</td>
<td>15</td>
</tr>
<tr>
<td>mELINMIntGetMessageTag</td>
<td>Reception</td>
<td>Returns the tag of a received message</td>
<td>16</td>
</tr>
<tr>
<td>mELINMIntGetRXPointer</td>
<td>Reception</td>
<td>Returns a pointer to a received message</td>
<td>17</td>
</tr>
<tr>
<td>mELINMIntRXMessageAvailable</td>
<td>Reception</td>
<td>Checks for any message arrival</td>
<td>18</td>
</tr>
<tr>
<td>mELINMIntRXStatus</td>
<td>Reception</td>
<td>Returns the status of a received message</td>
<td>19</td>
</tr>
<tr>
<td>mELINMIntRXErrorDetected</td>
<td>Reception</td>
<td>Checks for errors in reception</td>
<td>20</td>
</tr>
<tr>
<td>mELINMIntRXErrorTag</td>
<td>Reception</td>
<td>Returns the error tag</td>
<td>21</td>
</tr>
<tr>
<td>mELINMIntRXErrorCode</td>
<td>Reception</td>
<td>Returns the error code of a reception</td>
<td>22</td>
</tr>
<tr>
<td>mELINMIntCheckWakeUPReceived</td>
<td>Bus Control</td>
<td>Signals that a slave issued a wake-up</td>
<td>23</td>
</tr>
<tr>
<td>mELINMIntSendWakeUPSignal</td>
<td>Bus Control</td>
<td>Sends a wake-up signal to the slaves</td>
<td>24</td>
</tr>
<tr>
<td>mELINMIntSleepTimeOut</td>
<td>Bus Control</td>
<td>Signals when bus Idle time exceeded Sleep time-out</td>
<td>25</td>
</tr>
</tbody>
</table>
mELINMIntInitialize()

This macro initializes the driver.

Syntax
mELINMIntInitialize();

Parameters
None

Return Values
0 – Initialization OK
!= 0 – Error in initialization

Preconditions
None

Side Effects
None

Remarks
In this first version of the protocol, no error can be returned. However, in order to be compatible with future versions that may incorporate error returns, designers must include the proper test.

Example
if(mELINMIntInitialize()) // if an error in initialization was detected
{
    // error handling
}
else // if NO error (macro returned 0)
{
    // normal processing
}
mELINMIntTXBufferAvailable()

This macro checks if there is a transmission buffer available. The application must call this macro before trying to initiate any transmission.

Syntax
mELINMIntTXBufferAvailable();

Parameters
None

Return Values
1 – There is an available buffer to be used to transmit a message
0 – No buffer is currently available for transmission

Preconditions
The protocol must have been successfully initialized using the ELINMIntInitialize(void) function.

Side Effects
None

Remarks
None

Example
if(mELINMIntTXBufferAvailable())  // check if there is an available TX buffer
{
    // init transmission from this point
}
mELINMIntGetTXPointer()

This macro returns a pointer to the available transmission buffer.

**Syntax**

mELINMIntGetTXPointer(tag);

**Parameters**

*tag*  
The tag of a message. This is an identification of the message to be saved and used by the LIN protocol to inform the application of an eventual error that the transmission of a specific message suffered. In the event of an error being detected, the user can access the tag of the error message and with this tag, read the error code.

**Return Values**

**(BYTE *)** – A byte type data pointer to the transmission buffer. The application will load the message to be transmitted using this pointer.

**Preconditions**

1. The protocol must have been successfully initialized by the ELINMIntInitialize(void) function.
2. The mELINMIntTXBufferAvailable() macro must have been invoked with success.

**Side Effects**

None

**Remarks**

The total size of the message to be loaded must not exceed the maximum allowed size defined by ELINMINT_MAX_MESSAGE_SIZE (ELINMInt.def).

**Example**

```c
pt=mELINMIntGetTXPointer(3); // get the pointer to message #3
pt[0]=mymsg[0]; // insert first message byte
pt[1]=mymsg[1]; // insert second message byte
```
mELINMIntSendMessage(tag, i, s)

This macro requests the transmission of a message through LIN.

Syntax
mELINMIntSendMessage(tag, i, s);

Parameters
- **tag** The tag that identifies a message, previously defined by the application when calling the mELINMIntGetTXPointer macro.
- **i** The ID of the message, ranging from 0x00 to 0x3F. Bits 6 and 7 of the ID will be filled with parity bits and their original content ignored.
- **s** The size of the message, limited to 8 for all standard messages and to ELINMINT_MAX_MESSAGE_SIZE in the case of an extended frame (ID = 0x3E or ID = 0x3F).

Return Values
None

Preconditions
1. The mELINMIntTXBufferAvailable macro must have been successfully invoked.
2. The mELINMIntGetTXPointer macro must have been invoked.
3. The data buffer shall have been loaded with the message pointer.

Side Effects
None

Remarks
1. Calling this macro doesn’t ensure that the message will be successfully transmitted. The application must check the result of the transmissions by:
   - Waiting until the message transmission is completed (using mELINMIntMessageSent) and then checking if an error was detected in that message.
   - Checking if a transmission buffer is available (using mELINMIntTXBufferAvailable) and if an error is detected, evaluating which message (identified by its tag) presented a problem and the nature of the problem.
2. The ID = 0x3F is reserved by the LIN Consortium for future expansion (see LIN Specification Package 1.3, Protocol Specification Chapter 3.2) and therefore, it’s use may compromise future compatibility.
3. The macro takes the size of the message and calculates the minimum and maximum frame times to be used by the underlying function. If the size is passed in a variable, the calculations are done in real-time, requiring several cycles; however, if the application always calls this macro with fixed values instead of variables, then these calculations can be made in compile time, therefore, saving both code space and processing time.

Example 1 (Fixed Size)
mELINMIntSendMessage(tag, myID, 4); // requests transmission, size = 4, better!
while(mELINMIntMessageSent(tag)==0) // wait transmission to be completed
{
    if(mELINMIntTXStatus(tag)==ELINMINT_NO_ERROR)// check transmission status
    {
        // no error, normal processing
    }
    else // if error detected
    {
        // error handling
    }
Example 2 (Variable Size)

```c
mELINMIntSendMessage(tag, myID, msgSize); // requests transmission, variable size
while(mELINMIntMessageSent(tag)==0) // wait transmission to be completed
{
    if(mELINMIntTXStatus(tag)==ELINMINT_NO_ERROR)// check transmission status
    {
        // no error, normal processing
    }
    else // if error detected
    {
        // error handling
    }
}
```
mELINMIntTXStatus(tag)

This macro checks the status of a message already transmitted.

**Syntax**

mELINMIntTXStatus(tag);

**Parameters**

*tag*  This byte contains a message tag which is an identification of the message that was sent.

**Return Values**

The error code, defined according to the following table:

<table>
<thead>
<tr>
<th>#define</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINMINT_NO_ERROR</td>
<td>No error was detected</td>
</tr>
<tr>
<td>ELINMINT_THMIN_ERROR</td>
<td>Header time too short</td>
</tr>
<tr>
<td>ELINMINT_THMAX_ERROR</td>
<td>Header time too long</td>
</tr>
<tr>
<td>ELINMINT_TFMIN_ERROR</td>
<td>Frame time too short</td>
</tr>
<tr>
<td>ELINMINT_TFMAX_ERROR</td>
<td>Frame time too long</td>
</tr>
<tr>
<td>ELINMINT_CHECKSUM_ERROR</td>
<td>Checksum incorrect</td>
</tr>
<tr>
<td>ELINMINT_DATA_ERROR</td>
<td>Received and transmitted bytes don’t match</td>
</tr>
<tr>
<td>ELINMINT_FRAMING_ERROR</td>
<td>Framing error</td>
</tr>
</tbody>
</table>

**Preconditions**

The `mELINMIntSendMessage` macro must have been invoked.
The message transmission was completed, checked by `mELINMIntMessageSent`.

**Side Effects**

None

**Remarks**

This macro returns the result of the transmission of a specific message identified by *tag*.

**Example**

```c
mELINMIntSendMessage(9,0x04,2); // send a message
while(mELINMIntMessageSent(9)==0) // wait transmission message #9 completed
{
    if(mELINMIntTXStatus(9)==ELINMINT_NO_ERROR) // check transmission status
        // no error, normal processing
    }
else // if error detected
    // error handling
```
mELINMIntMessageSent(tag)

This macro checks if a message identified by tag was already sent.

Syntax
mELINMIntMessageSent(tag);

Parameters

*tag*  This byte contains a message tag which is an identification of the message, which the driver can use to track a specific message.

Return Values
1 – Message already sent
0 – Message not yet sent

Preconditions
The mELINMIntSendMessage macro must have been invoked.

Side Effects
None

Remarks
This macro flags when a specific message transmission is complete. However, it doesn’t ensure that the transmission was successful. The application must check it using mELINMIntTXErrorDetected.

Example
mELINMIntSendMessage(9,0x04,2); // send a message
while(mELINMIntMessageSent(9)==0) // wait transmission message #9 completed
;
mELINMIntTXErrorDetected()

This macro flags if an error was detected in the transmission of a message.

Syntax
mELINMIntTXErrorDetected();

Parameters
None

Return Values
1 – Error detected
0 – No Error detected

Preconditions
Called after detecting that a message was transmitted either by the mELINMIntMessageSent macro or by the mELINMIntTXBufferAvailable macro.

Side Effects
None

Remarks
None

Example 1
mELINMIntSendMessage(9,0x04,2); // send a message
while(mELINMIntMessageSent(9)==0) // wait transmission #9 to complete
{
    if(mELINMIntTXErrorDetected()) // check if an TX error was detected
    {
        errTx=mELINMIntTXErrorTag(); // if detected let's find the message
        // that caused the error
        // error handling
    }
}

Example 2
mELINMIntSendMessage(9,0x04,2); // send a message
while(mELINMIntTXBufferAvailable()) // wait for an available buffer
{
    if(mELINMIntTXErrorDetected()) // check if an TX error was detected
    {
        errTx=mELINMIntTXErrorTag(); // if detected let's find the message
        // that caused the error
        // error handling
    }
}
mELINMIntTXErrorTag()

This macro returns the tag of the message that presented an error.

Syntax

mELINMIntTXErrorTag();

Parameters

None

Return Values

A byte with the tag of the error message.

Preconditions

Error has been previously detected by mELINMIntTXErrorDetected().

Side Effects

None

Remarks

None

Example

errorTag=mELINMIntTXErrorTag();   // read the tag of the message that had an error
mELINMIntTXErrorCode(tag)

This macro returns in one byte the error associated with a message.

Syntax
mELINMIntTXErrorCode(tag);

Parameters
tag The identification of the message where an error was detected.

Return Values
The error code, defined according to the following table:

<table>
<thead>
<tr>
<th>#define</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINMINT_NO_ERROR</td>
<td>No error was detected</td>
</tr>
<tr>
<td>ELINMINT_THMIN_ERROR</td>
<td>Header time too short</td>
</tr>
<tr>
<td>ELINMINT_THMAX_ERROR</td>
<td>Header time too long</td>
</tr>
<tr>
<td>ELINMINT_TPMIN_ERROR</td>
<td>Frame time too short</td>
</tr>
<tr>
<td>ELINMINT_TPMAX_ERROR</td>
<td>Frame time too long</td>
</tr>
<tr>
<td>ELINMINT_CHECKSUM_ERROR</td>
<td>Checksum incorrect</td>
</tr>
<tr>
<td>ELINMINT_DATA_ERROR</td>
<td>Received and transmitted bytes don’t match</td>
</tr>
<tr>
<td>ELINMINT_FRAMING_ERROR</td>
<td>Framing error</td>
</tr>
</tbody>
</table>

Preconditions
1. Error detected by mELINMIntTXErrorDetected.
2. Tag of the related message read by mELINMIntTXErrorTag.

Side Effects
None

Remarks
None

Example
if(mELINMIntTXErrorDetected()) // check if an TX error was detected
{
    errorTag=mELINMIntTXErrorTag(); // get the tag of the message
    errorCode=mELINMIntTXErrorCode(errorTag); // find the error code
    // error handling
}
mELINMIntRXBufferAvailable()

This macro flags if there is a reception buffer available.

**Syntax**
mELINMIntRXBufferAvailable();

**Parameters**
None

**Return Values**
0 – No buffer available
1 – Buffer available

**Preconditions**
The protocol must have been successfully initialized by the ELINMIntInitialize(void) function.

**Side Effects**
None

**Remarks**
None

**Example**

```c
if (mELINMIntRXBufferAvailable()) // if there is a reception buffer available
    mELINMIntReceiveMessage(5,0x01,2); // request data: tag #5, ID=0x01 and Size=2
```
mELINMIntReceiveMessage(tag, i, s)

This macro requests a message to be sent from a slave.

Syntax

mELINMIntReceiveMessage(tag, i, s);

Parameters

tag The tag defined by the application to be associated with the incoming message requested.
i The ID of the requested message.
s The size of the message in bytes.

Return Values
None

Preconditions
Reception buffer available (detected by mELINMIntRXBufferAvailable).

Side Effects
None

Remarks
The request of reception, like in the transmission, doesn't ensure that the received message is correct; therefore, the application must check the results.

Example

mELINMIntReceiveMessage(5, 0x01, 2); // request data: tag #5, ID=0x01 and Size=2
mELINMIntMessageReceived(tag)

This macro checks if a message was received.

Syntax
mELINMIntMessageReceived(tag);

Parameters

Parameter | Description
---|---
tag | The received message identification.

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Message received</td>
</tr>
<tr>
<td>0</td>
<td>Message not yet received</td>
</tr>
</tbody>
</table>

Preconditions

Reception of a message previously requested by the mELINMIntReceiveMessage macro.

Side Effects

None

Remarks

This macro detects the reception of a message, but doesn’t ensure its correctness; therefore, the application must check it.

Example

```c
mELINMIntReceiveMessage(7,0x01,2); // request data:tag=5, ID=0x01, size=2
while(mELINMIntMessageReceived(7)==0) // wait this message to be received
    ;
```
mELINMIntGetMessageTag()

This macro returns the tag (identification) of a message that was received.

Syntax
mELINMIntGetMessageTag();

Parameters
None

Return Values
The tag (identification) of the received message.

Preconditions
1. The reception of a message must have already been requested using the mELINMIntReceiveMessage macro.
2. The message reception Acknowledged by the mELINMIntMessageReceived() macro.
3. No error detected as per the mELINMIntRXErrorDetected() macro (returning 0).

Side Effects
None

Remarks
None

Example
if(mELINMIntRXErrorDetected()==0) // check if an reception error was detected
{
    // if no error then
    Tag=mELINMIntGetMessageTag(); // get the tag of the message received
    pt=mELINMIntGetRXPointer(Tag); // get the data pointer of the message
}
mELINMIntGetRXPointer(tag)

This macro returns a pointer to the received message.

Syntax
mELINMIntGetRXPointer(tag);

Parameters

- **tag**  The tag identifying the message.

Return Values

- `(BYTE *)` – A byte type data pointer to the reception buffer. The application can read the message using this pointer.

Preconditions

1. Reception of a message detected by the `mELINMIntMessageReceived` macro.
2. No error detected by `mELINMIntRXErrorDetected` (return 0).
3. Tag of the message read using the `mELINMIntGetMessageTag` macro.

Side Effects

None

Remarks

None

Example

```c
Tag=mELINMIntGetMessageTag(); // get the tag of the message received
pt=mELINMIntGetRXPointer(Tag); // get the data pointer
```
mELINMIntRXMessageAvailable()

This macro checks for the reception of a message.

Syntax
mELINMIntRXMessageAvailable();

Parameters
None

Return Values
1 – Message Received
0 – No Message received

Preconditions

Side Effects
None

Remarks
Because the reception of a message is Acknowledged even when an error has occurred, the application will always check the integrity of the received message.

Example
if(mELINMIntRXMessageAvailable())  // check for received message
{
    //
}

mELINMIntRXStatus(tag)

This macro checks the status of a received message.

**Syntax**

mELINMIntRXStatus(tag);

**Parameters**

*tag*  
The identification of the message.

**Return Values**

The error code, defined according to the following table:

<table>
<thead>
<tr>
<th>#define</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINMINT_NO_ERROR</td>
<td>No error was detected</td>
</tr>
<tr>
<td>ELINMINT_THMIN_ERROR</td>
<td>Header time too short</td>
</tr>
<tr>
<td>ELINMINT_THMAX_ERROR</td>
<td>Header time too long</td>
</tr>
<tr>
<td>ELINMINT_TFMIN_ERROR</td>
<td>Frame time too short</td>
</tr>
<tr>
<td>ELINMINT_TFMAX_ERROR</td>
<td>Frame time too long</td>
</tr>
<tr>
<td>ELINMINT_CHECKSUM_ERROR</td>
<td>Checksum incorrect</td>
</tr>
<tr>
<td>ELINMINT_DATA_ERROR</td>
<td>Received and transmitted bytes don’t match</td>
</tr>
<tr>
<td>ELINMINT_FRAMING_ERROR</td>
<td>Framing error</td>
</tr>
</tbody>
</table>

**Preconditions**

3. Reception completed, checked with mELINMIntMessageReceived.

**Side Effects**

None

**Remarks**

None

**Example**

```c
if(mELINMIntRXMessageReceived(tag)) // check for received message
{  
  status=mELINMIntRXStatus(tag);   // if error handle it
  if(status)
  {
  
  }
  else // otherwise read it
  {
  
  }
}  
```
mELINMIntRXErrorDetected()

This macro checks for reception errors.

Syntax
mELINMIntRXErrorDetected();

Parameters
None

Return Values
1 – Error detect
0 – No error detect

Preconditions
3. Reception of message Acknowledged – mELINMIntMessageReceived.

Side Effects
None

Remarks
Because the reception of a message is Acknowledged even when an error has occurred, the application will always check the integrity of the received message with this macro. In case of an error, the mELINMIntRXErrorTag and mELINMIntRXErrorCode macros are called to identify the exact nature of the problem.

Example
if(mELINMIntRXErrorDetected()) // check if an RX error was detected
{
    // error handling
}
mELINMIntRXErrorTag()

This macro returns the tag of the message that presented an error.

Syntax
mELINMIntRXErrorCode();

Parameters
None

Return Values

tag The identification of the error message that was received.

Preconditions
An error detected by the mELINMIntRXErrorDetected macro.

Side Effects
None

Remarks
None

Example
if(mELINMIntRXErrorDetected()) // check if an RX error was detected
{
   ErrorTag=mELINMIntRXErrorTag(); // find the tag of the message with error
   //
}
mELINMIntRXErrorCode(tag)

This macro returns the code identifying the error detected in the reception of the message identified by tag.

Syntax

mELINMIntRXErrorCode(tag);

Parameters

tag    The identification of the error message received.

Return Values

The error code, defined according to the following table:

<table>
<thead>
<tr>
<th>#define</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINMINT_NO_ERROR</td>
<td>No error was detected</td>
</tr>
<tr>
<td>ELINMINT_THMIN_ERROR</td>
<td>Header time too short</td>
</tr>
<tr>
<td>ELINMINT_THMAX_ERROR</td>
<td>Header time too long</td>
</tr>
<tr>
<td>ELINMINT_TFMIN_ERROR</td>
<td>Frame time too short</td>
</tr>
<tr>
<td>ELINMINT_TFMAX_ERROR</td>
<td>Frame time too long</td>
</tr>
<tr>
<td>ELINMINT_CHECKSUM_ERROR</td>
<td>Checksum incorrect</td>
</tr>
<tr>
<td>ELINMINT_DATA_ERROR</td>
<td>Received and transmitted bytes don’t match</td>
</tr>
<tr>
<td>ELINMINT_FRAMING_ERROR</td>
<td>Framing error</td>
</tr>
</tbody>
</table>

Preconditions

1. Error detected by the mELINMIntRXErrorDetected macro.
2. The tag of the error read with the mELINMIntRXErrorTag macro.

Side Effects

None

Remarks

None

Example

ErrorCode=mELINMIntRXErrorCode(ErrorTag);// read the error code for the given tag
mELINMIntCheckWakeUPReceived()

This macro flags the reception of a wake-up signal from the slave.

Syntax
mELINMIntCheckWakeUPReceived();

Parameters
None

Return Values
1 – Wake-up received
0 – No wake-up received

Preconditions
Protocol initialized by the ELINMIntInitialize function.

Side Effects
None

Remarks
1. This macro flags the reception of a wake-up (reception of 0x80 from a slave). According to the specifications (LIN Specification Package 1.3, Twudel Parameter), the Master must wait at least a 4-bit time before starting communication. Therefore, once the application detects the wake-up signal, it must allow this minimum time before starting the communication process.

2. This signal is kept active until a message transmission or reception is completed.

Example
if(mELINMIntCheckWakeUPReceived())  // once received the wake-up from a slave
{                                   // process it to detect what happened
    
}
mELINMIntSendWakeUPSignal()

This macro sends a wake-up signal to the slaves.

Syntax

mELINMIntSendWakeUPSignal();

Parameters

None

Return Values

None

Preconditions

Protocol initialized by the ELINMIntInitialize function.

Side Effects

None

Remarks

This macro sends a wake-up signal (0x80). According to the LIN specifications *(LIN Specification Package 1.3, Twudel Parameter)*, the Master must wait at least a 4-bit time before starting communication. Therefore, once the application sends the wake-up signal, it must allow this minimum time before starting the communication process.

Example

mELINMIntSendWakeUPSignal();
mELINMIntSleepTimeOut()

This macro detects the time-out of the bus.

Syntax
mELINMIntSleepTimeOut();

Parameters
None

Return Values
1 – Time-out
0 – No time-out

Preconditions
None

Side Effects
None

Remarks
The time-out condition is detected when no bus activity is observed for a time interval larger than 25000 bits. In this
case a flag is set and the condition can be detected by calling this macro and if necessary, a wake-up signal may
be issued (mELINMIntSendWakeUPSignal).

Example
if(mELINMIntSleepTimeOut())  // if an sleep time-out is detected
{
    //
}

DRIVER USAGE

There are two ways to add the driver to a project:

1. Through Application Maestro™ Software:
   Select the module, adjust the parameters and select the directory location of the files that are going to be copied. Please refer to the Application Maestro documentation of this module for further explanation.
   After this, the designer must include the ELINMInt.c file in the project (inside MPLAB® IDE, as a C18 C Compiler source code file) and copy the following include file in all source code files accessing the protocol:
   
   \#include "ELINMInt.h"

2. Manually:
   To add the driver into the project, do the following:
   a) Copy all three files in the source code directory of the project.
   b) Include the ELINMInt.c file in the project (inside MPLAB® IDE, as a C18 C Compiler source code file).
   c) Copy the following include file in all source code files accessing the protocol:
      \#include "ELINMInt.h"
   d) Adjust the parameters of the driver. These parameters are located inside the ELINMInt.def file and are described in Appendix B: “ELINMInt.def Parameter Setup”.

Note: This first version of the protocol supports only one communication buffer, therefore, the use of tags wouldn’t be necessary. Future versions will implement multiple buffers (queuing), therefore, the application shall always send and receive the messages with the proper tag assignment.
Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the “Company”) is intended and supplied to you, the Company’s customer, for use solely and exclusively with products manufactured by the Company. The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

APPENDIX A: CODE EXAMPLE

```c
#include "ELINMInt.h" // .H file – to be added in all source files that reference LIN routines or constants
void InterruptVectorHigh(void); // prototype of the routines used in this test
void InterruptVectorLow(void);
void InterruptHandler(void);
void main(void);

BYTE my_msg[8]; // message buffer, for tests

/***************************************************************************/
/* Function: void Main(void) */
/* PreCondition: */
/* Input: */
/* Output: */
/* Side Effects: */
/* Stack Requirements: */
/* Overview:Main function of the LIN Master Test Firmware */
/***************************************************************************/

void main(void)
{
  char leds;
  unsigned int mydelay;
  BYTE *pt;
  BYTE ErrorCode;
  BYTE ErrorTag;
  BYTE Tag;
  BYTE rxtag;
  mydelay=2000; // initialize the delay variable
  TRISC=0x9F; // init serial pins - TX and control (LIN driver)
  PORTCbits.RC5=0; // negative edge (initial) pulse in the control
  if(mELINMIntInitialize()==0) // initialize Enhanced USART and LIN
    ErrorCode=mELINMIntInitErrorCode(); // if an error was detected return the Initialization
  T0CON=0xC8; // initialize TIMER0
  INTCON_TMR0IE=1; // enable timer0 int.
  INTCON_PIE1=1; // enable ints.
  PORTCbits.RC5=1; // positive edge pulse in the control pin (LIN Driver)
  INTCON_GIE=1;
  while(mydelay--)
    // initial delay - necessary to MCP201.
    // (Data Sheet - param. Tcsor)
```

while(1) // run forever
{
    Tag=0; // init Tag
    mydelay=600;
    while(mydelay--); // give a delay between messages, to make scope
    // visualization easier
    
    /******************************************************************************/
    // First receive a single message using fixed value tag
    // checking for the reception of that specific message
    /******************************************************************************/

    while(mELINMIntRXBufferAvailable()==0) // if there is no RX buffer available wait
        ;
    mELINMIntReceiveMessage(5,0x01,2); // request data using tag=5 (message number),
        // ID=0x01, size=2
    while(mELINMIntMessageReceived(5)==0) // wait until the message is received
        ;
    if((ErrorCode=mELINMIntRXStatus(5))) // check if an RX error was detected
        {
        leds++; // error handling - to be added at this point by
        application
        }
    else // otherwise (no error)
        {
        pt=mELINMIntGetRXPointer(5); // get the data pointer
        my_msg[0]=*pt; // read the message
        pt++;
        my_msg[1]=*pt;
        // received message handling - to be added at this
        // point by the application
        }
    mydelay=600;
    while(mydelay--); // give another delay
    
    /******************************************************************************/
    // Send a single message using fixed value tag
    // checking for the transmission of that specific message
    /******************************************************************************/

    while(mELINMIntTXBufferAvailable()==0) // Wait TX buffer available
        ;
    pt= mELINMIntGetTXPointer(3); // get available pointer and tag it's message as 3
    *pt=my_msg[0]; // insert data
    pt++;
    *pt=0;
    mELINMIntSendMessage(3,0x04,2); // send message
    while(mELINMIntMessageSent(3)==0) // wait until transmission message #3 completes
        ;
    if((ErrorCode=mELINMIntTXStatus(3))) // check if an TX error was detected (!=0)
        {
        leds++; // error handling - to be added at this point by
        application
        }
    mydelay=600;
    while(mydelay--); // give another delay
}
//*****************************************************/
// Check for Sleep Time-Out and wake-up if necessary
//*******************************************************************************/

if(mELINMIntSleepTimeOut()) // if timeout set (more than 25000 bit-time
{ // of silence in the BUS
    mELINMIntSendWakeUPSignal(); // send wake-up signal to the slaves
    // add application code here
}

//*****************************************************/
// Check for Wake-Up Signal sent by slave
//*******************************************************************************/

if(mELINMIntCheckWakeUPReceived()) // check for Wake-Up signals received
{
    if(mELINMIntSleepTimeOut()) // if timeout already set (more than 25000 bit-time)
    { // of silence in the BUS
    }
    else // if no timeout something unexpected happened, process
    {
    }
} // while (1)
} // void main(void)

//*******************************************************************************/
// High priority interrupt vector
#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh(void)
{
    _asm
        bra InterruptHandler // jump to interrupt routine
    _endasm
} //pragma code InterruptVectorHigh = 0x08
void InterruptVectorLow(void)
{
    _asm
        bra InterruptHandler // jump to interrupt routine
    _endasm
}

/*******************************************************************************/
* Function:    void InterruptHandler(void)
*               
* PreCondition: 
*               
* Input:       
*               
* Output:      
*               
* Side Effects: 
*               
* Stack Requirements: 
*               
* Overview:High priority interrupt routine 
*
#pragma code
#pragma interrupt InterruptHandler
void InterruptHandler(void) {
    if (INTCON_TMR0IF)
        ELINMIntHandler(); // process LIN int. based protocol
    TMR0L|=0x80; // timer0 int. every 128 counts - 128 instructions
                // time as timer0 is not using any prescaler (1:1)
    INTCON_TMR0IF=0; // reset int0 flag
}
APPENDIX B: ELINMInt.def PARAMETER SETUP

The ELINMInt.def file has many parameters, some of which must be adjusted by the designer. The parameters that require adjustment are:

CLOCK_FREQ

Description
This is the main clock frequency used by the microcontroller when running the driver.

Minimum 4000000L
Maximum 40000000L

Remarks
None

Example
#define CLOCK_FREQ16000000L // define the main clock as 16MHz

ELINMINT_BAUD

Description
This is the baud rate to be used in the bus by the LIN driver.

Minimum 1000L
Maximum 20000L

Remarks
None

Example
#define ELINMINT_BAUD19200L // Baud Rate adjusted to 19200 Baud

ELINMINT_MAX_MESSAGE_SIZE

Description
This is the maximum size of a message either to be transmitted or received.

Minimum 2
Maximum 255

Remarks
None

Example
#define ELINMINT_MAX_MESSAGE_SIZE16 // maximum size 16 bytes
ELINMINT_INTERRUPT_PERIOD

Description
This is the interrupt period of the interrupt routine (in microseconds) used by the application.

Minimum
The minimum period of this timer-based interrupt must be larger than the sum of the total time used by the LIN interrupt handler, plus the interrupt latency and any other task run by the application in the interrupt. The minimum time required by the LIN interrupt handler is calculated inside (ELINMin.H) as follows:

// number of instructions run by the protocol during an interrupt
#define ELINMINT_NINST_HANDLER_MIN112L

// here the necessary time to run the protocol during an int. is calculated as a function of the uC's clock frequency (CLOCK_FREQ) and the number of inst. of the LIN int. handler
#define ELINMINT_INT_HANDLER_TIME((1000000L*(4*(ELINMINT_NINST_HANDLER_MIN+5)))/CLOCK_FREQ)

// after that the processing time of the protocol is compared with the interrupt execution time and if smaller (interrupt period > processing time) then set an error message.
#if ELINMINT_INT_HANDLER_TIME>ELINMINT_INTERRUPT_PERIOD
    #error "LIN TIMING NOT VIABLE - INTERRUPT PERIOD TOO SMALL !!"
#endif

A safe approach is to assume that the handler is going to take about ELINMINT_NINST_HANDLER_MIN instructions, multiply by a safety margin value (e.g., 1.2) and calculate the time spent in microseconds (multiplying the result by (4/CLOCK_FREQ)).
If the interrupt process time becomes larger than the interrupt period, an error message will be issued and the compiling process will fail.

Maximum
8*(ELINMINT_BAUD)

Remarks
As a rule of thumb, a good interrupt period should be smaller than 3-bits time (3/ELINMINT_BAUD) because this time may affect the interbyte delay.

Example
#define ELINMINT_INTERRUPT_PERIOD 128L // 128usec interrupt rate
ELINMINT_INTERBYTE_SPACE

Description
This is the delay time added between the transmission of two bytes in a message. This delay time is automatically calculated based on the baud rate and the interruption period.

Minimum 0
Maximum

Remarks
In some systems, the slave requires a delay slightly larger than the one provided. In these cases, the designer may have to increase the size of the delay. It is recommended to increase this value in steps of one, as an excessive increment in the delays may lead to errors, like excessive header time or excessive frame time.

Example
With ‘1’ added to the original Interbyte space.

#define ELINMINT_INTERBYTE_SPACE ((ELINMINT_INTERBYTE_MIN+ELINMINT_INTERBYTE_MAX)/2 +1)
APPENDIX C: REFERENCES

- LIN Consortium (http://www.lin-subbus.de)
  The authoritative reference, the LIN Consortium provides all standards and specifications necessary to understand and implement LIN-based communication systems.

- Microchip (http://www.microchip.com)
  Microchip provides extensive support to LIN:
  - Hardware:
    Transceivers, LIN Enabled Microcontrollers, Development Boards
  - Software:
    Application Notes, Code Examples and Templates
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 intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Trademarks
The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeeLog, MPLAB, PIC, PICmicro, PICSTART, PRO MATE and PowerSmart are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
AmpLab, FilterLab, microID, MXDEV, MXLAB, PICMASTER, SEEVAL, SmartShunt and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.
Application Maestro, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICkit, PICDEM, PICDEM.net, PowerCal, PowerInfo, PowerMate, PowerTool, rflAB, rfPIC, Select Mode, SmartSensor, SmartTel and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.
All other trademarks mentioned herein are property of their respective companies.
© 2003, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.
 Printed on recycled paper.