LIN Protocol Implementation on the T89C51CC01/02

CAN Microcontrollers

Application Note

The Controller Area Network (CAN) has seen enormous success in automotive body and powertrain control systems. However, there is a change in emphasis arising in the automotive industry in which CAN is seen as too powerful and expensive for simple digital sensor/actuator body control applications. The emerging protocol Local Interconnect Network (LIN – Specification 2.0) has been proposed for such applications. The LIN protocol is introduced in this application note, along with its implementation on the T89C51CC01 or T89C51CC02. Both USART and bit manipulation methods are described for LIN Master and Slave drivers, whose C-source code is available from Atmel.

In-vehicle Network Applications

Today, electronic components and systems account for over 20% of the cost of a high-end passenger car. In-vehicle network applications are already successfully applied to body and powertrain control systems. This figure is expected to grow significantly over the next ten years with the introduction of increasingly more complex control systems such as Drive-by-Wire and multimedia systems giving access to the Internet. As far as in-vehicle control networking technology is concerned, an emerging growth area is in the adoption of a low cost body control sub-bus with which to complement CAN.

Many body control functions are often simple digital on/off operations such as activating lights, wipers, windows, etc. These are considered soft requirement real time systems that do not necessarily need the response that can be provided by CAN. Additionally, there is also the need in the industry to integrate a network node into a sensor or actuator. LIN is currently the candidate technology for these applications since it is a complementary technology to CAN but at much lower cost. Table 1 briefly compares CAN and LIN protocols.

Rev. 4189A–CAN–03/03

LIN Applications

The LIN protocol was introduced in 1999 as a low-cost sub-bus system to complement CAN in applications such as:

- **Vehicle Roof** (Rain Sensor, Light Sensor, Light Control, Sun Roof)
- **Vehicle Doors** (Mirror, Central Locking, Mirror Switch, Window Lift)
- **Engine** (Sensors, Small Motors, Steering Wheel, Cruise Control Switches, Wiper, Turn Signal, Radio, Climate Control)
- **Seat** (Seat Position Motors, Seat heater, Occupancy Sensor)

The main features of LIN are:

- Based on USART
- Physical Layer, enhanced ISO-9141
- Baud rate up to 20K baud
- 16 LIN ID, 8 Data Bytes
- Master/Slave communication

The application of LIN gives rise to the type of vehicle body control network architecture as described in Figure 1. Figure 1 shows an example LIN sub-bus for control of a car door electrical sub-system. The main body control CAN network interfaces with the rest of the door system via a LIN-CAN Bus Gateway Unit. A single LIN signal line is used to connect the LIN-CAN Bus Gateway Unit to the Smart Door Lock Unit, Smart Mirror Motor Unit and Smart Window Motor Unit. This would result in a total of three signal wires, which is at least five less than the traditional implementation. Therefore, the cost saving comes from the reduction in wiring offset by the additional cost of the three additional low-end microcontrollers.

LIN Protocol

A typical LIN frame is shown in figure 2 and consists of the following components:

Sync Break (LIN Master Task)

- At least 13 bits -Signifies start of frame

Sync Field (LIN Master Task)

- 1 bit - Sync Delimiter
- Hex 55 (8 bit data, start & stop bits) - Allows slaves to synchronize

ID Field (LIN Master Task)

- 8 bit data, start & stop bits
- Bits 0 to 3 - LIN ID
- Bits 4 to 5 - Length Control
- Bits 6 to 7 - Parity

Data Field (LIN Slave Task)

- 2, 4 or 8 bytes (each 8-bit data, start & stop bits)

Checksum 8 bits (LIN Slave Task)

The communication concept is that of Master and Slave tasks as shown in Figure 2 and Figure 3. The Master Task transmits the Message Header which consists of the Synch Break, Synch Field and Identifier Field. The Slave Task transmits Message Response which consists of the Data Field and Checksum Field. The Master Task is always transmitted by the Master Node, while the Slave Task can be transmitted by any of the nodes on the LIN subnet. The Master Node transmits the Slave Task when it is necessary to send data and receives via the Slave Task.

Table 1. Comparison of the Main Features of CAN and LIN

	CAN	**LIN**
Affiliation	Robert Bosch GmbH® CAN in Automation	The LIN Consortium
Maximum Baud Rate	1M baud - High Speed CAN ISO-11898	20K baud Enhanced ISO-9141 (ISO-K)
Network Topology	Bus	Bus
Network Access Method	Non-destructive Bit-wise Arbitration	Master initiates transmission, therefore no arbitration necessary
Number of Nodes	Usually limited by physical layer or higher layer protocol to 128 or 64	16 nodes (1 Master, 15 Slave)

Figure 1. Typical Use of a Combination of LIN Sub-bus Along with the CAN Bus in Localized Area of Car Door Control

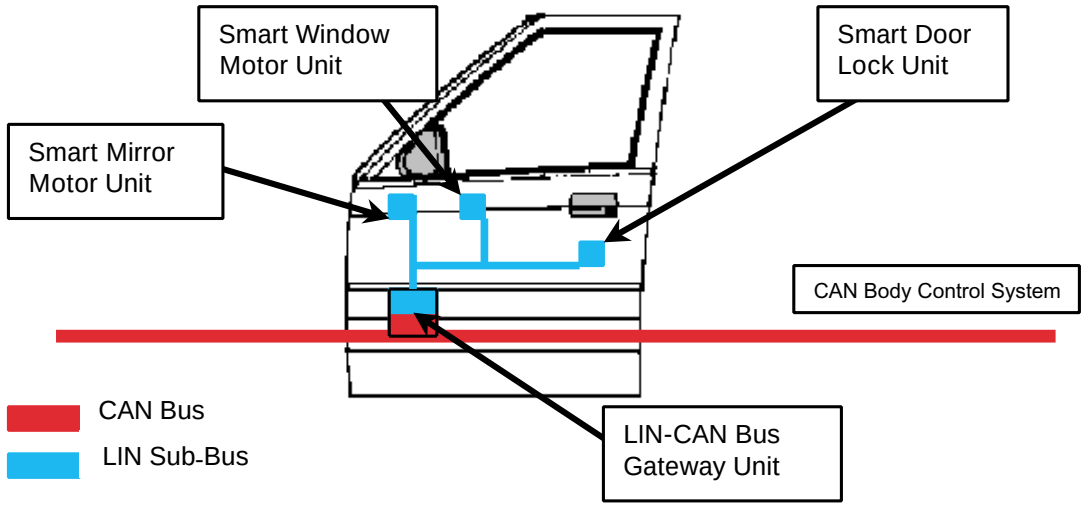
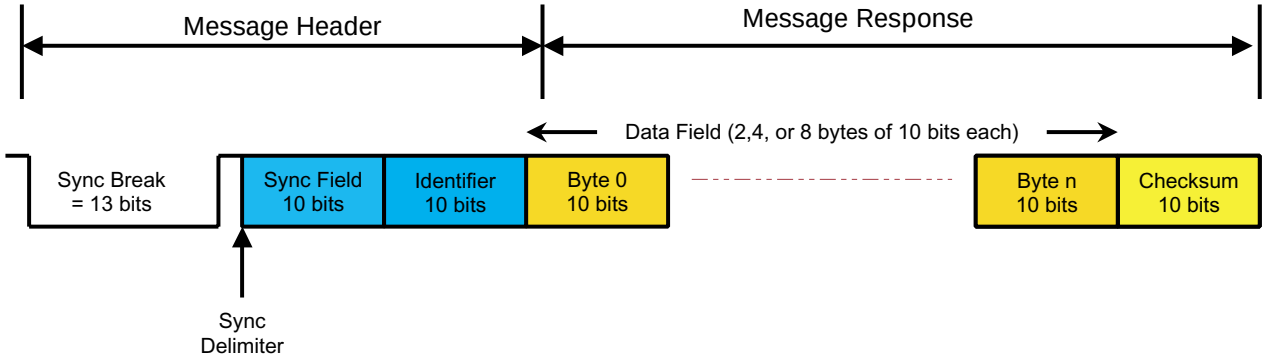
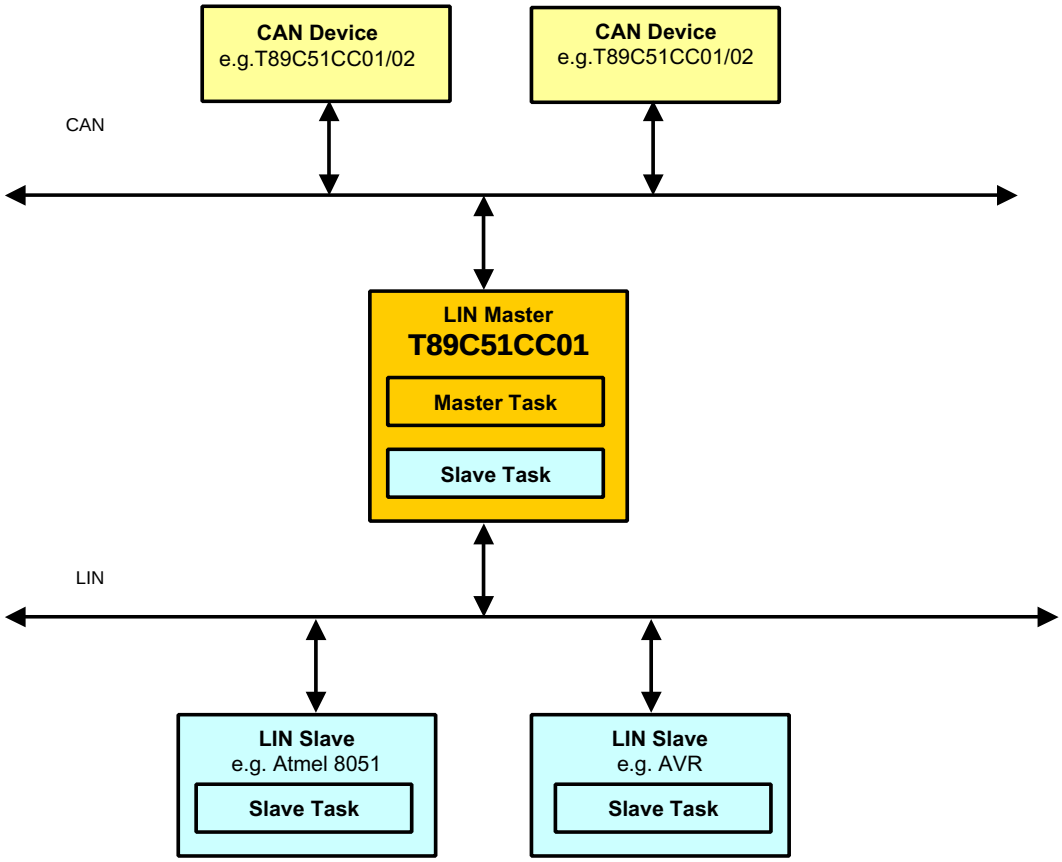


Figure 2. Typical LIN Frame Consisting of Sync Break, Sync Field, Identifier, Data Field and Checksum



Each byte of data follows a standard UART format of 10 bits (i.e., start bit, eight data bits, stop bit).

Figure 3. Typical Application of a T89C51CC01 Microcontroller as a CAN/LIN Gateway and a LIN Master Control Unit



Synch Break

The Synch Break signifies the beginning of a LIN frame and is always generated by the Master Task. The Synch Break tells the Slave Task to prepare for the Synch Field. The Synch Break consists of two different parts; a dominant (low) level that should be a minimum of 13 bit-times (Tbit) which is followed by a recessive (high) period that should be in the range one to four Tbit (Sync Delimiter).

The length of the Synch Field has been chosen to be at least 13 Tbit so that LIN Slave Tasks can distinguish between a valid Synch Break and the maximum possible allowed sequence of dominant bit within a data frame (i.e. 9 Tbit).

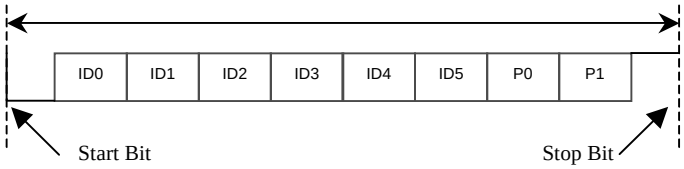
Synch Field

The Synch Field contains the signalling required for the slaves to synchronize with the Master's clock. The Synch Field is a byte containing the data "0x55" giving a waveform with five falling edges and five rising edges. These edges can be used during synchronization to tune the slave node transmission and reception speed to match the Master node. Therefore, the bit time is found by measuring the time of either 5 rising edges or 5 falling edges and dividing by eight (logical shift by 3 bits).

Identifier Field

The Identifier Field contains information about the contents and length of a message. This field is divided into three sections as shown in Figure 4; identifier bits (four bits), length control bits (two bits) and parity bits (two bits).

Figure 4. Identifier Field



The Identifier bits (ID0 to ID3) represent the identifier of the node who is to respond in the Message Response part of the frame. Thus there can be up to 16 nodes on a LIN subnet; one master and up to 15 slaves. The Identifier Field therefore describes the content of the message and not the destination.

Table 2. Number Bytes in Data Field as Dictated by Length Control

ID5	ID4	NData (No. of Bytes)
0	0	2
0	1	2
1	0	4
1	1	8

The final two bits in the Identifier Field are parity bits which are defined by a mixed parity algorithm. This ensures that the Identifier Field will never consist of an all Dominant or all Recessive bit pattern. It must be noted that the parity check only detects errors, it does not correct them.

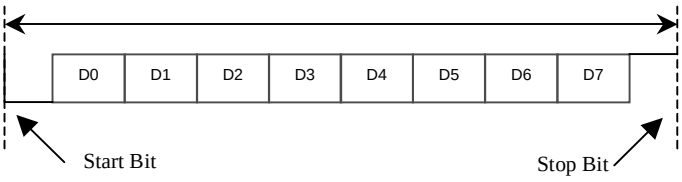
The parity check bits are calculated by the following mixed parity algorithm:

$\overline{P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4}$
$P1 = ID1 \oplus ID3 \oplus ID4 \oplus ID5$

Data Field

The Data Field contains two, four or eight bytes of data, each consisting of one stop bit, eight data bits and one stop bit. Transmission is done with the LSB first. The Data Field is written by the responding slave task. Since there is no bus arbitration, only one slave task should be allowed to respond to each identifier. All other slave tasks are limited to reading the response and act accordingly. The format of a single byte within the Data Field is shown in Figure 5.

Figure 5. A Single Byte of the Data Field

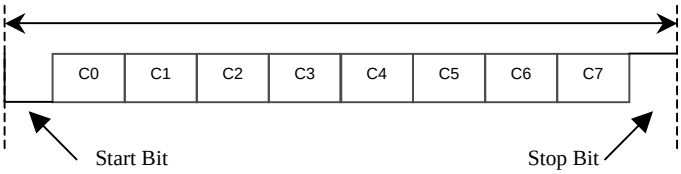


Checksum Field

The last field in the Message Frame is the Checksum Field. This byte contains the inverted modulo-256 sum of all data bytes within the Data Field. The sum is calculated by doing an “Add with carry” on all data bytes and then inverting the answer. The properties of the inverted modulo-256 sum are such that if this number is added to the sum

of all data bytes, the result will be “0xFF”. The format of the Checksum Field is shown in Figure 6.

Figure 6. Checksum Field



It is important that the carry bit is added to the LSB during the addition of each byte in the “Add with Carry” operation so that:

0xFF + 0x01 = 0x01

Error Detection

The following three methods of error detection are outlined in the LIN specification:

- **Bit Error** detection is concerned with any unit that is sending a bit on the LIN bus monitors that bit. A **Bit Error** occurs when a monitored bit value is different to the bit value sent.
- **Checksum Error** detection of the Data Field as described previously.
- **Identifier Parity Error** detection for the Identifier Field as described previously.
- **Slave Not Responding Error** detection is concerned with ascertaining when a Slave Task does not respond to a Master Task.
- **No Bus Activity Error** detection is concerned with ascertaining when a valid Synch Break has not been received for a certain time period.
- **Inconsistent Synch Field Error** detection is concerned with ascertaining when the edges of the Synch Field go outside of tolerance.

The responsibility for detection of these errors is split between Master and Slave nodes.

Error Signalling and Recovery Time

The direct signalling of errors is not possible due to the master concept. Errors should be locally detected and provided in the form of diagnostics on request.

Fault Confinement

The LIN specification specifies that the Master node should handle most (if not all) of error detection, error recovery and diagnostics. Fault confinement depends upon the system requirements and therefore only basic support is specified. Therefore much of the fault confinement needs to be specified by the user particularly for their applications needs.

Master Control Unit Fault Confinement

The Master shall detect the following error situations:

- **Bit Error** or **Identifier Parity Error** in the Master Task
- Slave Task in Master Control Unit -> **Slave Not Responding Error** and **Checksum Error**.

The Master Control Unit Fault Confinement is not specified in LIN specification v1.2. However, a recommended practice is outlined. The recommended practice states that a *Master Task* sending data must detect a **Bit Error** in the Synch or Identifier Fields. The Master Control Unit should keep track of any transmission error by incrementing the *MasterTransmitErrorCounter*. It is increased by 8 each time a bit in the Synch or Identifier Fields are locally corrupted.

It is decreased by 1 (not less than 0) each time both Fields are read back correctly. If the counter is incremented beyond C_MASTER_TRANSMIT_ERROR_THRESHOLD (recommended default value = 64) it is assumed a massive disturbance has occurred on the LIN bus and an error handling procedure should be taken in the application code level.

The *Slave Task* in the Master Control Unit sending data will detect **Bit Errors**. The *Slave Task* in the Master Control Unit receiving data will detect either **Slave Not Responding Error** or a **Checksum Error**. When a **Bit Error** occurs, *MasterTransmitErrorCounter* is incremented as described previously. When **Slave Not Responding Error** or **Checksum Error** is detected when expecting or receiving data, *MasterReceiveErrorCounter* is increased by 8. It is decreased by 1 (not less than 0), each time the slave response is received correctly with no errors. If *MasterReceiveErrorCounter* is incremented beyond C_MASTER_RECEIVE_ERROR_THRESHOLD (recommended default value = 64), it is assumed that the addressed slave has malfunctioned and an error handling procedure should take place in the application code level.

When a **Slave Not Responding Error** (i.e. detected by the Master Control Unit) occurs, the LIN specification v1.2 does not specify whether or not retransmissions should take place. This is a decision that should take place within the application code level.

Slave Control Unit Fault Confinement

The Slave node should detect the following error situations:

- **Bit Error** detection when transmitting data via Slave Task.
- **Checksum Error** when receiving data via the Slave Task.
- **Identifier Parity Error** when receiving the Message Header from the Master node.

In addition to these errors, two other errors should be detected but are dependent upon the application of the specific LIN drivers:

- **Slave Not Responding Error** should be detected when a Slave node is expecting data to come from another Slave node.
- **Inconsistent Synch Field Error** should be detected when the received Synch Field is out of tolerance.

The Slave Control Unit Fault Confinement is not specified in LIN specification v1.2. However, a recommended practice is outlined which is similar to the Master Control Unit's method. It is not outlined to the same extent as the Master Control Unit. However, a suggested method is outlined here, based on that of the Master Control Unit.

The suggested practice is that a *Slave Task* in a Slave Control Unit sending data must detect a **Bit Error** in the Data or Checksum Fields. The Slave Control Unit should keep track of any transmission error by incrementing the *SlaveTransmitErrorCounter*. It is increased by 8 each time a bit in the Data or Checksum Fields are locally corrupted. It is decreased by 1 (not less than 0) each time both Fields are read back correctly. If the counter is incremented beyond C_SLAVE_TRANSMIT_ERROR_THRESHOLD (recommended default value = 64) it is assumed a massive disturbance has occurred on the LIN bus and an error handling procedure must be taken in the application code level.

The *Slave Task* in the Slave Control Unit receiving data will detect a **Identifier Parity Error**, **Slave Not Responding Error** or a **Checksum Error**. When **Slave Not Responding Error** or **Checksum Error** is detected when expecting or receiving data, *SlaveReceiveErrorCounter* is increased by 8. It is decreased by 1 (not less than 0), each time the slave response is received correctly with no errors. If *SlaveReceiveErrorCounter* is incremented beyond C_SLAVE_RECEIVE_ERROR_THRESHOLD (recommended default value = 64), it is assumed that the addressed slave has malfunctioned and an error handling procedure should take place in the application code level.

Reserved Identifiers

Command Frame Identifier: Two COMMAND Frame Identifiers are reserved to broadcast general command requests for service purposes from the Master to the Slaves. Each frame has 8 bytes distinguished by the following reserve Identifiers; “0x3C” Master Request Frame and “0x7D” Slave Response Frame. The Master Request Frame is used to send commands and data to the slave nodes. The Slave Response Frame triggers on slave node (addressed by a prior download frame) to send data to the Master node.

If D7 of the 1st data byte of the Data Field is zero, then the frames usage is reserved for definition by the LIN consortium, otherwise the usage is free for custom definition.

Sleep Mode Command: The Sleep Mode Command is a broadcast message to put all slave nodes into sleep mode. The Sleep Mode Command is a Command Frame from the Master with the 1st byte in the Data Field as 0x00.

Extended Frames: A frame with Identifier Field as “0xFE” is reserved for the user defined message formats. A frame with Identifier Field as “0xBF” is reserved for future definition by the LIN consortium.

LIN Implementation on T89C51CC01

The T89C51CC01 is an 8051 compatible microcontroller and is ideal for implementation of a CAN to LIN gateway application. The implementation of a CAN driver on this microcontroller is described in another Atmel application note “CAN Bus Drivers for C51 Products”.

In a typical CAN and LIN application (see Figure 3), the T89C51CC01/02 is an ideal candidate to be a CAN/LIN gateway unit and therefore the LIN Master. However, for other applications, it may be required to use the T89C51CC01 as a slave instead. Therefore there are three driver types which are described and implemented in this application note:

- LIN Slave node using bit manipulation methods.
- LIN Master node using the USART peripheral.
- LIN Master node using bit manipulation (if for example, the USART had to be used for a different purpose).

Each of these methods will be demonstrated in C-source code examples available from Atmel and described further in the following sections of this application note.

The drivers contain routines via an Application Programming Interface for generation and handling of the LIN protocol and include the following error detection:

- Bit Error detection
- Checksum Error detection
- Identifier Parity Error detection

The following features of the LIN protocol are left for the user to develop specifically for their application due to current flexibility in the LIN specification:

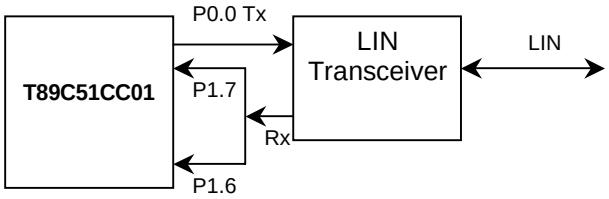
- Slave Not Responding Error
- No Bus Activity Error
- Inconsistent Synch Field Error
- Extended Frame processing
- Sleep Mode processing

The LIN C-source code drivers provide the basic support to enable the user to implement a LIN device on an T89C51CC01/02 and to be able to develop their own fault confinement strategy specifically for their applications needs.

LIN Slave Driver: Bit Manipulation

The LIN slave implementation uses the PCA function to synchronize with the LIN master. The bit manipulation method uses parallel port pins to be the Rx and Tx lines connecting to the LIN transceiver. To implement the Rx line, port pins that can detect positive and negative transitions of the LIN signal are required for a bit manipulation implementation (see Figure 7). Any of the digital port pins can be used to be the Tx line to the LIN transceiver (e.g. P0.0 is used in the C-source code examples).

Figure 7. Hardware Connections between T89C51CC01 and LIN Transceiver



Pins P1.6 and P1.7 are used for detection of positive and negative transition. Since each pin can only be configured for either positive or negative transitions, two pins must be electrically connected together as shown in Figure 7.

Bit Timing Configuration

The bit timing is derived from the received Synch Field using the on chip PCA. Timer 2 is used to sample the bits received and to generate the bits for transmission on the LIN bus.

C-Source Files

The C-source code drivers contain the following source files:

- LINSLAVE.C – LIN Slave protocol driver routines
- LIN.H – Header file for LINSLAVE.C
- MAIN.C – Function main()
- TIMER0.C – Scheduler routines
- TIMER.H – Header file for TIMER.C
- 89C51CC0.H – T89C51CC01/02 register definitions

LIN Driver Application Programming Interface

The API of the bit manipulation LIN Slave driver associated with this application note contains the following functions and global variables.

- **LIN_Init** (void) – Initializes the LIN driver.
- API Variables:
- *Checksum* – (Unsigned char) checksum to be sent or received checksum.
- *LinData[8]* – (unsigned char) the checksum is calculated from this buffer. This buffer also contains data either to be sent or received across the LIN bus.
- SlaveControlUnitId – (unsigned char) stores the LIN address of the Slave Control Unit.
- *LinStatusRegister* – (unsigned char) Stores Bit Error detection and other errors.
- Nbytes (unsigned char) – number of bytes in the LIN Data Field.

- *SlaveTransmitErrorCounter* – (unsigned char) stores the current value of transmission errors.
- *SlaveReceiverErrorCounter* – (unsigned char) stores the current value of received errors.

LIN Status Register is shown in Table 3

Table 3. LIN Status Register in the API

MSB							LSB
R	R	NBA	ISFE	BE	SNRE	CE	IPE

Where:

R	Reserved
NBA	No Bus Activity
ISFE	Inconsistent Sync Field Error
BE	Bit Error
SNRE	Slave Not Responding Error
CE	Checksum Error
IPE	Identifier Parity Error

Example Use of LIN Driver API

See Appendix A for code examples. The only hooks into the system that the programmer needs to set up are:

- Main() in main.c: Mainly for system initialisation.
  - Set up processor X2 mode
  - Set up processor ports
  - Initialize LIN address & variables
  - Initialize scheduler
  - Wake up LIN transceiver
- Timer0_ISR() in timer0.c: Mainly for reading sensors and populating the LinData buffer.

Code Size & Processor Loading

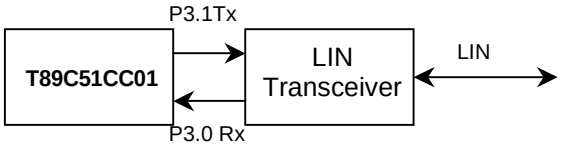
Using the Keil™ 8051 compiler, the code size is 1.5K bytes including main() and scheduler function in TIMER0.C.

Processor loading will depend upon baud rate implemented and frequency of tasks for reading/writing to sensors and actuators. However under tests for baud rates 2400, 4800, 9600 and 19200 using a 20 MHz crystal (the maximum value for theT89C51CC01), processor loading was found to be in the region of 38 to 40%.

LIN Master Driver: USART Peripheral

The USART implementation of the LIN protocol is the most efficient way of implementing a Master Control Unit since the USART contains much of the hardware support for generation of LIN bytes, i.e. start bit, 8-data bits, stop bit format. To implement a LIN Master Control Unit with the T89C51CC01, at least two port pins are required as shown in Figure 8.

Figure 8. Hardware Connections between T89C51CC01 and LIN Transceiver



Bit Timing Configuration

Bit timing for the USART is achieved using Timer 1 via register TH1. Timer 1 is used in mode 1 which is auto-reload of TH1, therefore generating the baud rate. Appendix D describes how the bit timing is set and gives example baud rates for 12 MHz, 16 MHz and 20 MHz crystals.

Master Task Implementation

The first problem to solve for LIN Master Driver implementation using a USART is that of generation of the Synch Break. As previously mentioned, this must be at least 13 bit times with the bus at dominant. The maximum number of bits that can be driven to dominant with the T89C51CC01 USART is 10 bits; one start bit followed by nine data bits at dominant by writing 0x000 to the USART (note that the stop bit is recessive).

Therefore, to generate the Synch Break of at least 13-bit times, a different strategy is used. The Synch Break is generated by changing the value in TH1, so that the baud rate is effectively decreased. Therefore writing 0x00 to the USART, will result in at least 13 bits times sent to dominant.

Once the Synch Break is generated, the baud rate is set to the full value again for transmission of the Synch and Identifier Fields. Bit Error detection is carried out upon each bit transmitted.

Slave Task Implementation

The Slave Task for transmission of the Data Field transmits the data at full baud rate. The Data Field is followed by transmission of the Checksum Field. Bit Error detection is carried out upon each bit transmitted.

Slave Task reception is carried out by initializing the USART interrupt and using an interrupt service routine to get each byte from the LIN bus.

C-Source Files

The C-source code drivers contain the following source files:

- LIN.C – LIN protocol driver routines
- LIN.H – Header file for LIN.C
- MAIN.C – Function main()
- TIMER0.C – Scheduler routines
- TIMER.H – Header file for TIMER.C
- 89C51CC0.H – T89C51CC01/02 register definitions

LIN Driver Application Programming Interface

The API of the USART LIN Master driver associated with this application note contains the following functions and global variables.

- **LIN_SetBaud** (unsigned char) – Set baud rate and must be called with one of the following macros (macros are defined in LIN.H):
  - BAUD_2400
  - BAUD_4800
  - BAUD_9600
  - BAUD_19200

- **LIN_ProcessMsg** (void) – Processes LIN message and schedules Master and Slave Tasks. The following variables are used by this function:
  - *LinData[8]* – (unsigned char) LIN Data Field information. This must be set when the LIN master is transmitting data. When LIN master is receiving data, this buffer will be filled with this information.
  - *LinStatusRegister* – (unsigned char) Stores Bit Error detection and other errors
  - *MasterControlUnitId* – (unsigned char) stores the LIN address of the Master Control Unit.
  - *MasterTransmitErrorCounter* – (unsigned char) stores the current value of transmission errors
  - *MasterReceiverErrorCounter* – (unsigned char) stores the current value of received errors
- **LIN_InitIdField** (void) – Generate Identifier Field contains both LinId & LenCtrl and automatically generates parity bits P0, P1. The following must be set:
  - LinId (unsigned char)
  - NBytes (unsigned char)
- **LIN_CalcChecksum** (void) – Calculates the Checksum from LinData[8]. The following variables are used by this function:
  - *Checksum* – The calculated checksum is stored in here.
  - *LinData[8]* – The checksum is calculated from this buffer.

LIN Status Register is shown in Table 4:

Table 4. LIN Status Register in the API

MSB							LSB
R	R	NBA	ISFE	BE	SNRE	CE	IPE

Where:

R	Reserved
NBA	No Bus Activity
ISFE	Inconsistent Sync Field Error
BE	Bit Error
SNRE	Slave Not Responding Error
CE	Checksum Error
IPE	Identifier Parity Error

Example Use of LIN Driver API

See Appendix B for code examples. The only hooks in the system that the programmer needs to set up are:

- Main() in main.c: Mainly for system initialisation.
  - Set up processor X2 mode
  - Set up processor ports
  - Initialize LIN variables
  - Set LIN baud rate

  - Initialize timer 0 (used for baud rate generation)
  - Wake up LIN transceiver
- Timer0_ISR() in timer0.c: Mainly for LIN message scheduling.
  - Set up LIN message schedule
  - Set LIN ID
  - Set up number of bytes to be sent
  - If Master is to transmit data, set up LIN Data buffer and calculate checksum
  - Initiate Message Transmission

Code Size & Processor Loading

Using a Keil 8051 compiler code size is approximately 1K byte including the main() and scheduler function in TIMER0.C.

Processor loading will depend upon the frequency of tasks for reading/writing to sensors and actuators, and the frequency of initiation of LIN frame transmission.

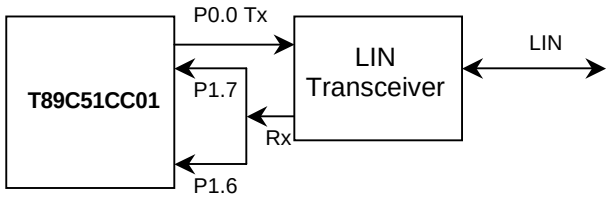
CPU load for the T89C51CC01 LIN driver was measured on a T89C51CC01 development board with 16 MHz crystal. X2 mode was used for maximum processing power. Tests were carried out at 19200 baud, 8 data bytes within the data field, which results in at least 124 bits per frame. Note that the maximum transmission period of a LIN frame of this size and data rate is approximately 6.5 ms (e.g. 52 µs x 124 bits = 6448 µs = 6.448 ms). The table below shows some example LIN frame transmission initiation periods versus% CPU loading:

LIN Frame Transmission Initiation Period	% CPU Load
100 ms	~0.51%
20 ms	~1.9%
10 ms	~2.83%

LIN Master Driver: Bit Manipulation

A bit manipulation method for Master Control Unit implementation would be required in the situation when the USART peripheral is unavailable for use. The bit manipulation method uses parallel port pins to be the Rx and Tx lines connecting to the LIN transceiver. To implement the Rx line, port pins that can detect +ve and –ve transitions of the LIN signal are required for a bit manipulation implementation (see Figure 9). Any of the digital port pins can be used to be the Tx line to the LIN transceiver (e.g. P0.0).

Figure 9. Hardware Connections between T89C51CC01 and LIN Transceiver



Pins P1.6 and P1.7 are used for detection of positive and negative transition. Since each pin can only be configured for either positive or negative transitions, two pins must be electrically connected together as shown in figure 11.

Bit Timing Configuration

The bit timing is derived from Timer 0 interrupt. Support is given in the source code for 2,400, 4,800, 9,600 and 19,200 baud for 16 MHz and 20 MHz crystal frequencies. However, the user can easily calculate the appropriate values for the Timer 0 registers for their crystal frequency. The timer clock rate is $F_{osc}/12$ in standard mode and $F_{osc}/6$ in X2 mode. The code written for this application note is for F_{osc} = 16 MHz or F_{osc}, both in X2 mode. Refer to the T89C51CC01 documentation and the LIN C-source for more information on how to set up the Timer 0 registers.

LIN Master Task Implementation

This is when the device is configured as a LIN Master device. A state machine and timer ISR are used to generate the Sync Break, Sync Field and Identifier Field. Once these fields are completed, control goes to the slave task.

LIN Slave Task Implementations

If the LIN ID transmitted by the Master Task is the same as the LIN Master ID, then the LIN Master device will execute a Slave Transmission Task and transmit the Data Field and Checksum Field. Otherwise the LIN Slave Receive Task procedure will be used to receive data from the LIN slaves.

For the LIN Slave Transmission Task, a state machine and timer ISR are used to generate the bits of the Data Field and Checksum Field. Once these fields are completed, the LIN message transfer is complete.

For the LIN Slave Receive Task, a sample point is set and a timer ISR samples the bus at regular intervals until the full number of bytes in the Data Field, and Checksum Field have been received.

C-Source Files

The C-source code drivers contain the following source files:

- LIN.C – LIN protocol driver routines
- LIN.H – Header file for LIN.C
- MAIN.C – Function main()
- TIMER0.C – Scheduler routines
- TIMER.H – Header file for TIMER.C
- 89C51CC0.H – T89C51CC01/02 register definitions

LIN Driver Application Programming Interface

The API of the Bit Manipulation LIN Master driver associated with this application note contains the following functions and global variables.

- **LIN_SetBaud** (unsigned char) – Set baud rate and must be called with one of the following macros (macros are defined in LIN.H):
  - BAUD_2400
  - BAUD_4800
  - BAUD_9600
  - BAUD_19200
- **LIN_InitVars** (void) – Initialises LIN driver
- **LIN_ProcessMsg** (void) – Processes LIN message and schedules Master and Slave Tasks. The following variables are used by this function:
  - *InstigateLinTx* – Bit (True = send message)
  - *LinData[8]* – LIN Data Field information. This must be set when the LIN master is transmitting data. When LIN master is receiving data, this buffer will be filled with this information.

  - *LinStatusRegister* – Stores Bit Error detection and other errors.
  - *MasterControlUnitId* – Unsigned char, stores the LIN address of the Master Control Unit.
  - *MasterTransmitErrorCounter* – Unsigned char, stores the current value of transmission errors
  - *MasterReceiverErrorCounter* – Unsigned char, stores the current value of received errors
- **LIN_InitIdField** (void) – Generate Identifier Field contains both LinId & LenCtrl and automatically generates parity bits P0, P1. The following must be set:
  - LinId (unsigned char)
  - NBytes (unsigned char)
- **LIN_CalcChecksum** (void) – Calculates the Checksum from LinData[8]. The following variables are used by this function:
  - *Checksum* – The calculated checksum is stored in here.
  - *LinData[8]* – The checksum is calculated from this buffer.

LIN Status Register is shown in Table 5:

Table 5. LIN Status Register in the API

MSB							LSB
R	R	NBA	ISFE	BE	SNRE	CE	IPE

Where

R	Reserved
NBA	No Bus Activity
ISFE	Inconsistent Sync Field Error
BE	Bit Error
SNRE	Slave Not Responding Error
CE	Checksum Error
IPE	Identifier Parity Error

Example Use of LIN Driver API

See Appendix B for code examples. The only hooks in the system that the programmer needs to set up are:

- Main() in main.c: Mainly for system initialisation.
  - Set up processor X2 mode
  - Set up processor ports
  - Initialize LIN variables
  - Set LIN baud rate
  - Initialize timer 0 (used for baud rate generation)
  - Wake up LIN transceiver
- Timer0_ISR() in timer0.c: Mainly for LIN message scheduling.
  - Set up LIN message schedule
  - Set LIN ID
  - Set up number of bytes to be sent
  - If Master is to transmit data, set up LIN Data buffer and calculate checksum
  - Initiate Message Transmission

Code Size & Processor Loading

Using the Keil 8051 compiler, the code size is 1.5K bytes including main() and scheduler function in TIMER0.C.

Processor loading will depend upon baud rate implemented and frequency of tasks for reading/writing to sensors and actuators. However under tests for baud rates 2400, 4800, 9600 and 19200 using a 20 MHz crystal (the maximum value for the T89C51CC01), processor loading was found to be in the region of 38 to 40%.

LIN Device Analysis and Conformance Testing

There are a number of test and analysis products available on the market. At the time of writing this application note, no conformance test was officially available. As a result, conformance testing has become the responsibility of the OEM whose role is the integration of LIN devices, possibly from different manufacturers. In general, the main features of LIN that need particular attention are:

- Parity and checksum generation (are they generated and interpreted correctly)
- Oscillator tolerance (LIN slave device - how much tolerance in)
- Sync Break tolerance
- Sync Delimiter tolerance

Summary and Conclusions

The T89C51CC01 and T89C51CC02 are ideal candidates for implementation of a CAN/LIN gateway. The LIN driver implementation is shown in this application note and the associated source code, while the CAN implementation is shown in the application note “CAN Bus Driver for Atmel C51 Products”.

A bit manipulation implementation has the advantage that it can self-synchronize from the Sync Field transmitted by the Master Task, but has the disadvantage of taking up more program memory than the USART implementation. Also, since the byte format is implemented in software, the bit manipulation suffers more from bit timing variability problems. These become more apparent at higher baud rates and longer physical LIN bus implementation. Therefore, bit manipulation is recommended only when the USART is unavailable for implementation.

A bit manipulation method for a LIN slave is ideal since the device can synchronize from the Synch Field transmitted by the Master Control Unit.

A USART implementation of the LIN Slave has not been shown in this application note or provided in the C-source code LIN driver examples. A LIN Slave Control Unit is required to synchronize from the Synch Field transmitted by the Master Control Unit, possibly using a low tolerance oscillator. The USART is not able to do this. However, for some applications in which the oscillator variability is not a problem, users may find it possible to use the USART for a LIN Slave Control Unit.

The code supplied is written entirely in the C programming language. Although efforts have been made to implement this efficiently, performance enhancements can be achieved by replacing C-code with in-line assembler. This has been left for users to carry out specifically for their application.

Abbreviations Used

API	Application Programming Interface
CAN	Controller Area Network
LIN	Local Interconnect Network
OEM	Original Equipment Manufacturer
PCA	Programmable Counter Array
USART	Universal Synchronous/Asynchronous Receiver Transmitter