XAPP432 (v1.1) April 3, 2007

Implementing a LIN Controller on a CoolRunner-II CPLD

Summary

LIN, or Local Interconnect Network, is a simple single-wire serial communications protocol designed primarily for use in automotive applications. Compared to CAN, LIN is a simpler and slower protocol, but its simplicity makes it ideal for decentralized sensor or display nodes, where CANbus introduces more complexity than is often necessary for low-speed monitoring or display purposes. This application note describes an implementation of a LIN controller on a Xilinx CoolRunner-II™ CPLD. A microcontroller interface is provided, but this could also be implemented as an IP core with minimal effort. See the PicoBlaze application note for details.

Introduction

LIN Controller Functionality

The LIN controller discussed in this document functions in a manner very similar to a UART. The controller handles all of the serialization, bit-level timing, frame packing, checksum generation and validation, parity generation and validation, and ensures and validates the consistency of the LIN frames. The physical interface is handled by an external LIN transceiver, as shown in Figure 1, available through a number of manufacturers.

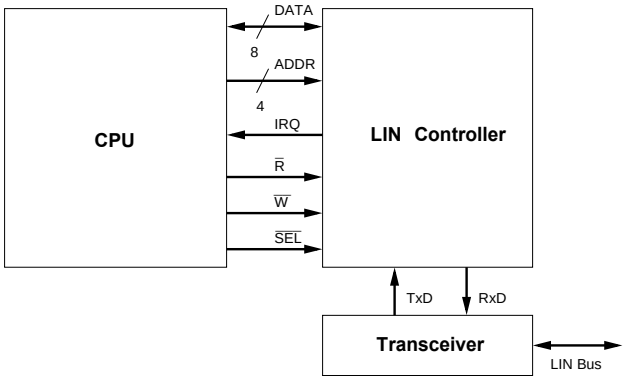

Figure 1: **CPU Interface and LIN Transceiver Interface.**

The CPU application interfaces with the controller through a set of addressable registers and an 8-bit wide data bus, as shown in Figure 1. The controller can interrupt the microcontroller on any LIN error condition or on reception or successful transmission of a single character. The interrupt is level-sensitive--the interrupt line will remain high until the error condition is acknowledged and cleared by the microcontroller.

Appropriate Uses for LIN

LIN is appropriate in any application that needs to be cost-effective, requires relatively low bitrates, and does not require robust fault management or reliability; in general, LIN is a good choice for events and applications that happen in "human time." In this capacity, it is an ideal protocol to handle communication between simple, noncritical components in an automobile.

For instance, a LIN node may control the dashboard display in an automobile. A typical automobile display only needs to be updated a few times a second, and requires only minimal redundancy and fault tolerance. Another LIN node may take input from a keypad. Others may be used to for in-car environmental monitoring, automatic door locks, or low-speed actuators such as intermittent wiper blades (Figure 2). LIN may also find uses in the growing popularity of luxury features, such as in-car navigation systems, in-car DVD players or entertainment systems, and individual per-seat environmental or entertainment controls.

LIN is not intended for use in safety applications such as airbag controllers, ABS, anti-skid systems, critical engine management components, or any other system that could affect the safety of a vehicle. If safety and robust fault tolerance is an issue, another protocol such as CAN should be considered instead.

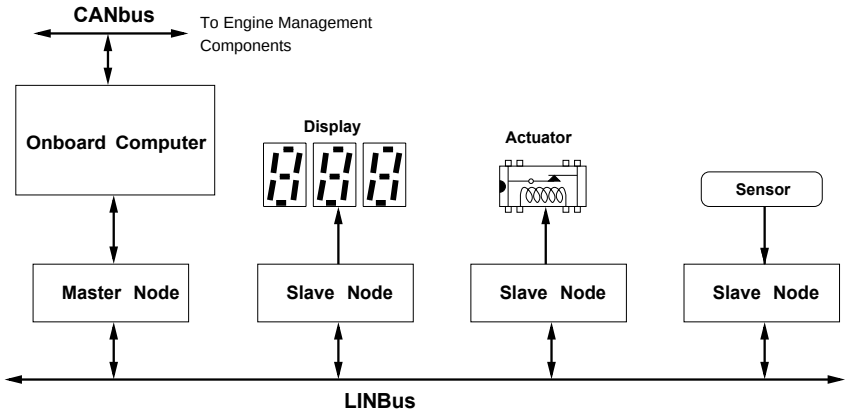

Figure 2: **Master and Slave Nodes in a Typical Application.**

CoolRunner-II Advantages for LIN

The Xilinx CoolRunner-II is a low-cost, low-power, programmable logic device that is ideal for this type of application. Used as a standalone controller, the CoolRunner-II fits well into the existing automotive design culture. A typical sensor or display node, for instance, may be comprised of a number of different technologies designed by groups that may only have minimal interaction or communication with each other. A typical airbag controller in an automobile today, for instance, may have one or two ASICs, a CANbus controller, a microcontroller, and various sensors all developed by different outsourced groups. In this type of development environment, flexibility and predictable, well-documented interaction between the various components is a must. A LIN controller implemented on a CPLD meets all of these criteria.

Changes in the LIN protocol could be implemented quickly with no impact on the overall design, footprint, protocol, or even in the software interacting with the LIN controller. LIN is an emerging standard, and as such may change in ways that would require the redesign of an ASIC used in the same role. If LIN is to gain widespread acceptance as a complementary protocol to CANbus, implementors must be assured that changes to the LIN protocol can be absorbed quickly with minimal impact to their prototypes.

Xilinx CPLDs are extremely flexible, finding applications in such diverse areas as LCD drivers, decoders and encoders, bus interfaces such as SPI or I^2C, conventional UARTs, and general I/O expansion. The LIN controller designed in this document uses approximately 80% of a 256 macrocell CoolRunner-II device, when optimized for space. The Xilinx CoolRunner-II family includes devices up to 512 macrocells, more than enough to implement a fully functional LIN node entirely within a single device. For example, the CPU interface could be removed, some minimal control and timing logic added, and an LCD or VFD driver could be added to create a LIN display node, all at a lower cost and in a footprint no larger than most microcontrollers.

The Xilinx CoolRunner-II family features extremely low power consumption--the lowest power CPLDs in the industry. For example, a Xilinx XC2C256 operating at 20MHz in a configuration utilizing approximately 80% of available resources, with 12.5% of the available flip-flops toggling per clock, consumes less than 10 mA of current, far less than that of similar CPLDs from other manufacturers.

LIN Protocol

LIN Consortium

The authoritative source of information on the LIN protocol is the LIN consortium. Their website, http://www.lin-subbus.org, has the latest LIN specification, as well as a forum, development and testing tools, and information

LIN vs. CAN

LIN is designed to be a complementary protocol to CAN. As the CAN protocol grows and develops, so does its complexity. The LIN specification can be implemented with a low part count and relatively low cost, and is ideal for low-speed multiplex displays, sensors or other inputs, and actuators. CAN, on the other hand, is robust, fault-tolerant, and extremely high-speed; thus, it is ideal for critical communications, such as engine management, airbags, or skid protection systems. However, it requires a relatively higher cost and higher part count. Table 1 provides a comparison between LIN and CAN.

Table 1: **LIN Compared to CAN**

Feature	LIN	CAN
Error Handling	Typical errors are specified and detectable using checksum, parity, and bus transmission monitoring, but retransmission, back off, or baud adjustments are not specified and are left to the upper-layer application.	Robust fault confinement and signalling, including global and local error detection, burst error detection, CRC error checking, fault confinement procedures, and retransmission.
Acknowledgement	No, left to upper-layer application.	Yes
Electrical Interface	2-valued bus (dominant and recessive). A 2-wire bus is specified in great detail, specifying slope, slew rates, allowable voltage and current ranges, and complex impedance.	The CAN electrical interface is not specified, only suggested. Suggestions are single-wire, differential, optical, etc.
Power Management	Sleep and wake-up are specified	Sleep and wake-up are specified
Bit Rates	Up to 19200 bps.	Up to 128 kbps or 1 Mbps, depending on the implementation and underlying physical bus.
Master/Slave	Single master, no arbitration.	Multimaster, arbitration.
Multicast	Yes	Yes
Synchronization	Yes	Yes

Table 1: **LIN Compared to CAN**

Feature	LIN	CAN
Prioritization	None	Yes
Frame Length	Fixed, depending on identifier: 2,4 or 8 bytes recommended. Optionally more to meet specific needs.	Variable from 0 to 8 bytes.

LIN Network and Frame Format

A LIN network is composed of a single master control unit (master node) and a number of slave control units (slave nodes). The master node contains both a master task and a slave task. Each slave node contains a single slave task, as shown in Figure 3.

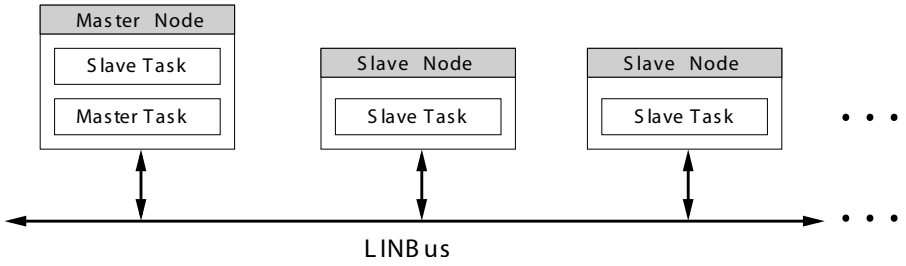

Figure 3: **LIN Network**

The master task is responsible for sending out a synch break, a one-byte synchronization field, and a one-byte identifier field. A single slave in the network will then respond to that identifier with 2, 4, or 8 data bytes, followed by a single-byte checksum. It is important to note that an identifier addresses a function, not a specific slave node. A LIN network is inherently multicast; consequently, every slave node on a network may receive and act on a message; however, only a single slave may respond to a given identifier. It is the responsibility of the LIN application to ensure that only a single slave responds on a given identifier. A maximum of 64 unique identifiers exist; out of these, 4 are reserved for future use. The number of total nodes in a network is limited by the maximum allowable network impedance, and should not exceed 16. A sample LIN transaction showing the division between the master and slave tasks is shown in Figure 4.

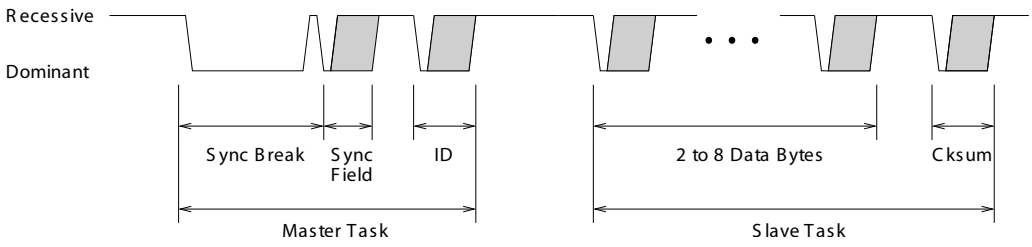

Figure 4: **LIN Transaction**

A LIN frame consists of a one-byte identifier, 2, 4, or 8 data bytes, and a one-byte checksum. A LIN byte starts with a single dominant start bit, 8 data bytes, LSb first, and a single recessive stop bit, as shown in Figure 5.

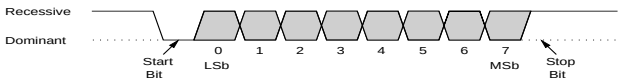

Figure 5: **LIN Byte Format**

The identifier contains 2 parity bits and a 6-bit identifier. Two bits of the 6-bit identifier optionally denote the expected length of the frame, as shown in Table 2.

Table 2: **Frame Length as a Function of the Identifier's Length Code**

Bit 5	Bit 4	Length (in bytes)
0	0	2
0	1	2
1	0	4
1	1	8

Implementation

Figure 6 gives a block diagram of the overall design of the LIN controller. Each functional block has a corresponding synthesizable VHD file in the ZIP file accompanying this document.

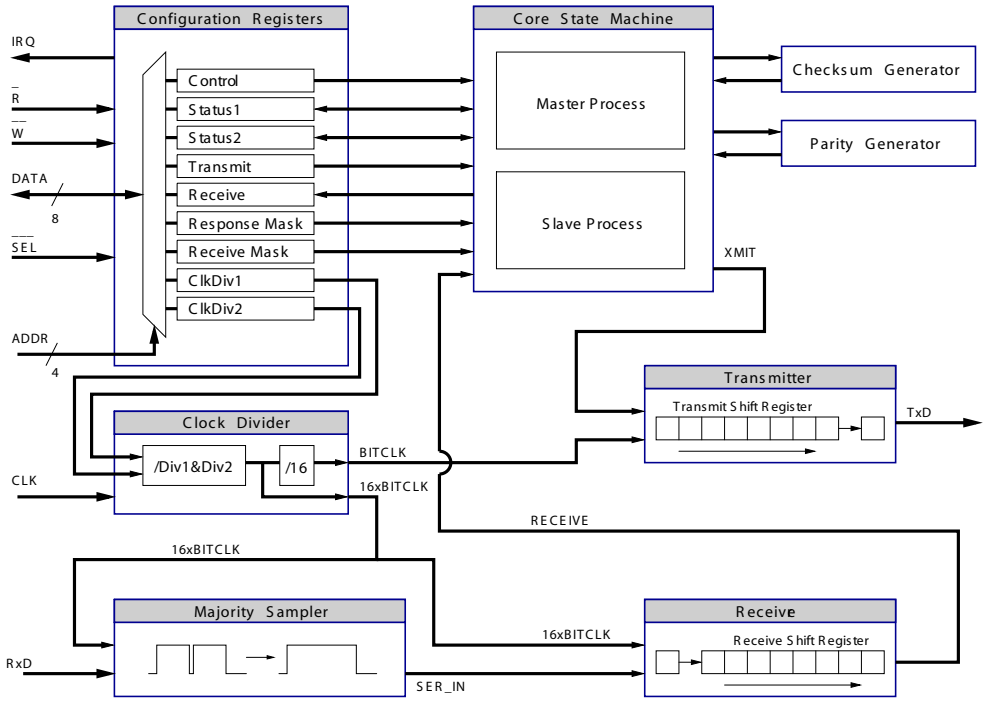

Figure 6: **Block Diagram for the LIN Controller and CPU Interface.**

Configuration Registers/CPU Interface

The configuration register block handles the bulk of the CPU interface, and mediates the setting and clearing of various status and configuration registers. The CPU-side interface acts as a set of addressable 8-bit registers, some of which can be cleared and set by the CPU, and others that can only be set or cleared by the controller itself. The register sets and their functions are listed in Table 3.

Table 3: **Register Sets and Their Functions.**

Tag	Addr.	Bit(s)	Name	R/W	Dflt.	Description
CONTROL1	0000	7	I_RECEIVE	R/W	1	0 - RDR full int. disabled 1 - RDR full int. enabled
		6	I_TRANSMIT	R/W	0	0 - TDR empty int. disabled 1 - TDR empty int. enabled
		5	I_ERROR	R/W	1	0 - Error interrupt disabled 1 - Error interrupt enabled
		4-2	Unused	-	-	-
		1	CFG_AUTOBAUD	R/W	0	Unimplemented
		0	CFG_MASTERSLAVE	R/W	0	0 - Controller is a master node. 1 - Controller is a slave node.
STATUS1	0010	7-4	Unused	-	-	-
		3	S_ERROR	R	0	0 - No error flags set 1 - An error flag is set Cleared when STATUS2 is read
		2	S_IDLE	R	0	0 - Controller is active. 1 - Controller is idle.
		1	S_TDRE	R	1	0 - TDR is full 1 - TDR is empty
		0	S_RDRE	R	1	0 - RDR is full 1 - RDR is empty

Table 3: **Register Sets and Their Functions.**

Tag	Addr.	Bit(s)	Name	R/W	Dflt.	Description
STATUS2	0011	7	Unused	-	-	-
		6	E_SHORTFRAME	R	0	1 - A short frame was received. Cleared when STATUS2 is read
		5	E_NOSLAVERESP	R	0	1 - No slave response within the time-out. Cleared when STATUS2 is read
		4	E_XMIT	R	0	1 - Bit error during transmission. Cleared when STATUS2 is read
		3	E_FRAMING	R	0	1 - A framing error was encountered. Cleared when STATUS2 is read
		2	E_PARITY	R	0	1 - A parity error was detected in an ID. Cleared when STATUS2 is read
		1	E_CKSUM	R	0	1 - Checksum error in a received frame. Cleared when STATUS2 is read
		0	E_OVERRUN	R	0	1 - RDR overrun - byte lost. Cleared when STATUS2 is read
TRANSMIT	0100	7-0	TDR	W	00	Transmit Data Register
RECEIVE	0101	7-0	RDR	R	00	Receive Data Register
ID_MASK	0110	7-0	ID_MASK	W/R	00	Slave ID Mask
ID_FILTER	0111	7-0	ID_FILTER	W/R	00	Slave ID Filter
CLKDIV1	1000	3-0	CLKDIV1	W/R	00	Clock Divisor, MSB
CLKDIV2	1001	7-0	CLKDIV2	W/R	02	Clock Divisor, LSB

Core State Machine

The core state machine handles all bit timing operations, sets or resets various status registers, and interrupts the CPU when appropriate. The core state machine is primarily composed of two state machines, the master and the slave. It also contains a counter for all bit timing operations, a break detection process to detect whether a break has been initiated, and a byte counter, which counts the expected number of bytes in a frame based on the identifier, as shown in Table 2.

Master Task

The master process is only active if the CFG_MASTERSLAVE flag is set. There should only be one master active in any single LIN network. The master task handles all bus arbitration--this is arbitration only in a very limited sense--the master task must initiate any slave response and must arbitrate and initiate any slave-slave communication.

Functionally, the master task sends out a synch break, a synch field, and a one byte slave identifier. The master task then goes idle and the slave task takes over and waits for or sends out any slave response.

The master task's behavior is shown in the flowchart in Figure 7.

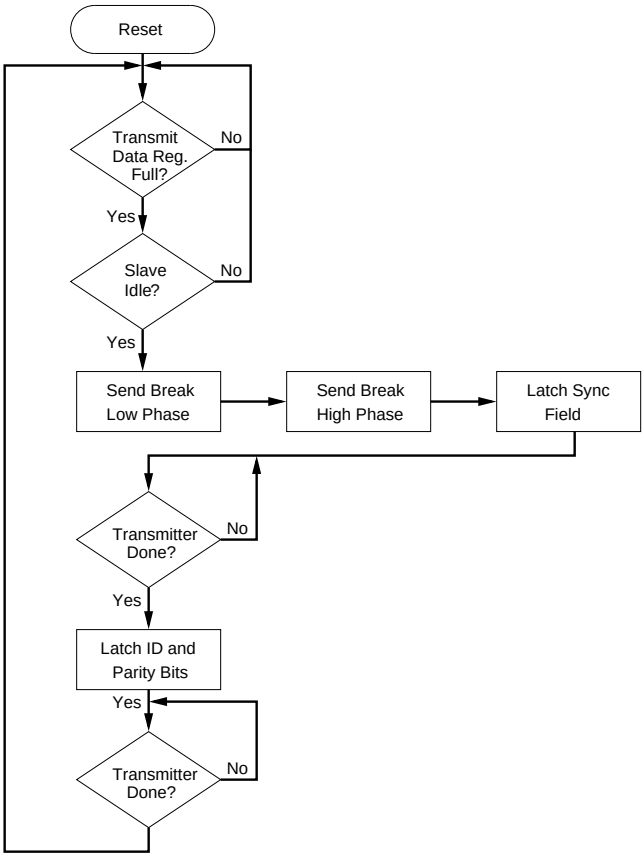

Figure 7: **Master Task Flowchart.**

Slave Task

The slave task is responsible for sending the data portion of a frame. If the slave is being addressed by the master (indicated by a match in its receipt mask), it will respond or take action on the contents of the message. There is NO procedure for slave arbitration, so it is the responsibility of the upper-layer application to ensure that only one slave responds on a given identifier, even if many slaves take other action on a given identifier.

The slave, functionally, waits for a synch break, synchronizes its internal clock based on the timing in the synch field sent out by the master, receives an identifier, and then, depending on

the mask and the upper-layer application, ignores the ID, responds to the message, or receives a message from another slave. A flowchart of the slave task's behavior is given in Figure 8.

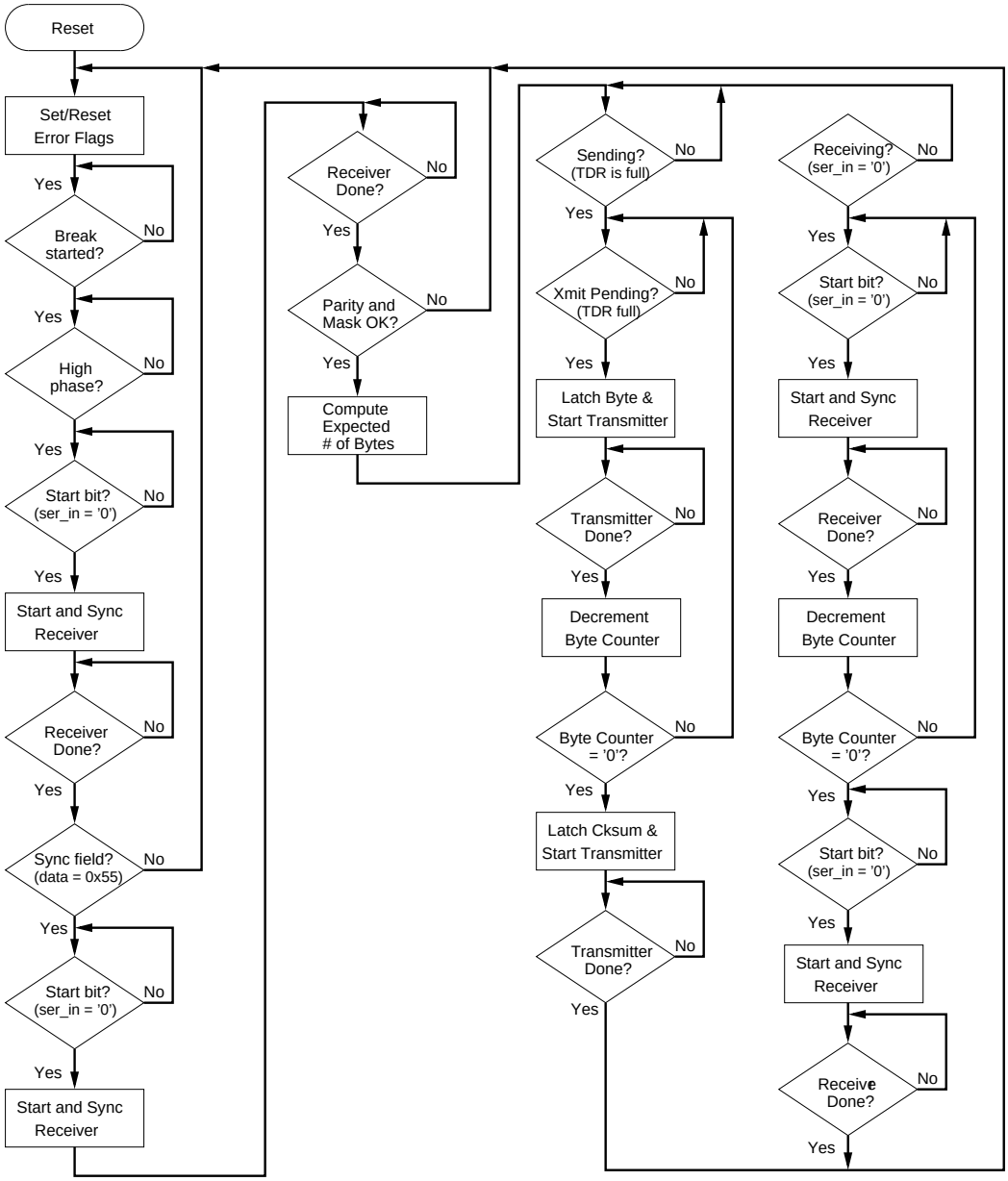

Figure 8: **Slave Task Flowchart.**

The LIN controller has a configurable ID mask and filter. The mask is first bitwise ANDed with the received ID, then compared to the filter. If the masked ID and filter do not match, the frame will be completely ignored (including any framing, checksum, or other errors) until the next synchronization break.

Filtering provides a way to reduce computational load on slave nodes. A slave may be configured to acknowledge only a small range of identifiers. The master node should not, in general, utilize any mask or filter, because if the mask indicates a frame should be ignored, the master could incorrectly assume the bus is idle and send a synchronization break while another node is in the process of transmitting. Further, if a frame is ignored, any errors associated with that frame will not be flagged. In general, the master node should receive all frames, and the upper layer application should assume responsibility for any additional filtering.

Clock Divider

The clock divider divides the system clock down to the internal clock "bitclk_x16" with a configurable 16-bit divisor. The MSB of this divisor is in register CLKDIV1, and the LSB is in CLKDIV2. This internal clock is then further divided by 16, generating an internal "bitclk," used for most bit timing operations. To compute the clock divisor given the system clock rate and the desired bit rate, use the following formula (& denotes concatenation):

$$divisor = \frac{CLK}{16 \times (desiredbitrate)}$$

$$\text{divisor} = \text{CLKDIV1 \& CLKDIV2}$$

For example, given a system clock frequency of 1.8432Mhz and a desired bit rate of 9600bps, the divisor is:

$$divisor = 12_{10} = \frac{1843200}{16 \times (9600)}$$

$$divisor = 12_{10} = 000C_{16}$$

$$CLKDIV1 = 00_{16}$$

$$CLKDIV2 = 0C_{16}$$

Divisors of 1 or 0 are invalid; the behavior is undefined if the divisor is 1 or 0.

The LIN specification requires that slave nodes have the ability to adjust their local clocks using the synchronization field sent out by the master. The ability to adjust the slave clock is not currently part of this implementation--each node must have a local resonator with an error no more than 1.5% relative to the master resonator, and must have divisors that match appropriately.

Table 4 lists divisors and errors for some common clock rates.

Table 4: **Clock Divisors for Common Bitrates**

1.8432 MHz clock			3.072 MHz clock			40.000 MHz clock		
Bitrate	**Divisor**	**Error**	**Bitrate**	**Divisor**	**Error**	**Bitrate**	**Divisor**	**Error**
2400	48	0.000%	2400	80	0.000%	2400	1042	-0.032%
9600	12	0.000%	9600	20	0.000%	9600	260	0.160%
19200	6	0.000%	19200	10	0.000%	19200	130	0.160%

Majority Sampler

The majority sampler samples the incoming bit stream at 16x the current bit rate. It outputs the majority of the last 16 samples --that is, if at least 8 of the last 16 samples are '1', the output is '1', otherwise it is '0'. This results in a delay between actual input (RxD) and the input seen by the internal state machines (ser_in) of 0.5 * BIT_TIME.

Checksum Generator

The checksum generator maintains a resettable accumulator, an 8-bit input, a "strobe" to latch and add a new value, and an 8-bit output containing the checkbyte (bitwise complement of the checksum). The checksum is computed as a running sum of all data bytes received, with any carry added back to the LSb of the sum. The checkbyte, transmitted over the bus, is the bitwise complement of the checksum.

The checksum generator does not add the carry until the next strobe; thus, to compute the correct checksum, a final "00" must be latched.

Parity Generator

The parity generator is a simple combinational circuit that takes a 6-bit input and outputs two parity bits PR0 and PR1. The parity generation and verification is part of LIN's error handling--it is used for simple integrity checking on the identifier field sent out by the master. The slave will raise a flag when an inconsistent parity is detected, but otherwise makes no distinction between an inconsistent parity and an unknown identifier that does not match its response or reception masks.

The parity bits are computed as follows (ID(0) is the LSb of the identifier field):

$$PR0 = ID(0) \oplus ID(1) \oplus ID(2) \oplus ID(4)$$

$$PR1 = \overline{ID(1) \oplus ID(3) \oplus ID(4) \oplus ID(5)}$$

Receiver

The receiver is implemented as a simple shift register with a parallel output and an asynchronous reset. The receiver internally divides the bitclk_x16 signal down to its own bitclk, enabling synchronization with the center of the bit being received. The receiver is asynchronously reset by the core state machine with a synchronized reset signal, which resynchronizes the internal bitclk generator and starts reception of a new byte. The receiver outputs an error if a framing error was encountered (incorrect stop bit) and outputs a "done" signal when byte reception is finished.

Transmitter

The transmitter module is implemented as a simple shift register with a parallel input and a strobe to load a new input. It signals "done" when the shift register has shifted out all bits and outputs an error if the last bit transmitted differs from the bit received (indicative of some sort of bus failure or transient error).

LIN Transceiver

The LIN transceiver simulates the behavior of a LIN transceiver and models the LIN bus. The bus is modeled as a weak-high, pulled low. The transceiver has one output (RxD), one input (TxD), and the LIN bus itself, which is bidirectional. This module is used for simulating a multiple-node LIN network.

CPLD Utilization

The LIN controller described in this document was targeted for the XC2C256 CoolRunner-II devices and uses approximately 80% of its resources when optimized for space. The utilization summary is given in Table 5.

Table 5: **XC2C256 Utilization**

Macrocells Used	Product Terms Used	Registers Used	Pins Used	Function Block Inputs Used
199/256 (78%)	696 /896 (78%)	168/256 (66%)	20 /118 (17%)	505/640 (79%)

CPU Interfacing

A standard CPU interface is provided for the controller. The CPU interface has a 4-bit wide address bus, a 8-bit wide bidirectional data bus, ground-true select, and separate ground-true read and write lines.

Register Access Timing

The registers are designed to be used asynchronously; that is, the CPU and the LIN controller can be in different clock domains, if proper setup and hold times are observed. The interface is

similar to a conventional UART. Figure 9 and Figure 10 show timing diagrams for register operations.

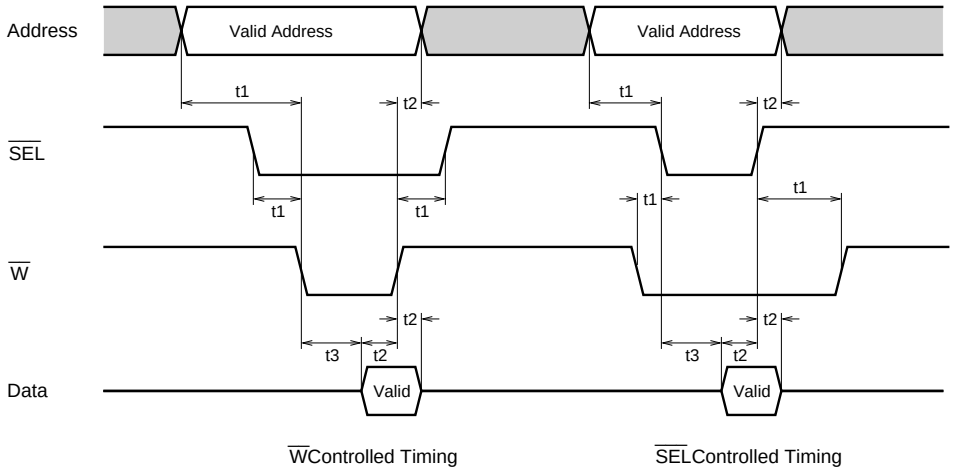

Figure 9: **Write Timing Diagram**

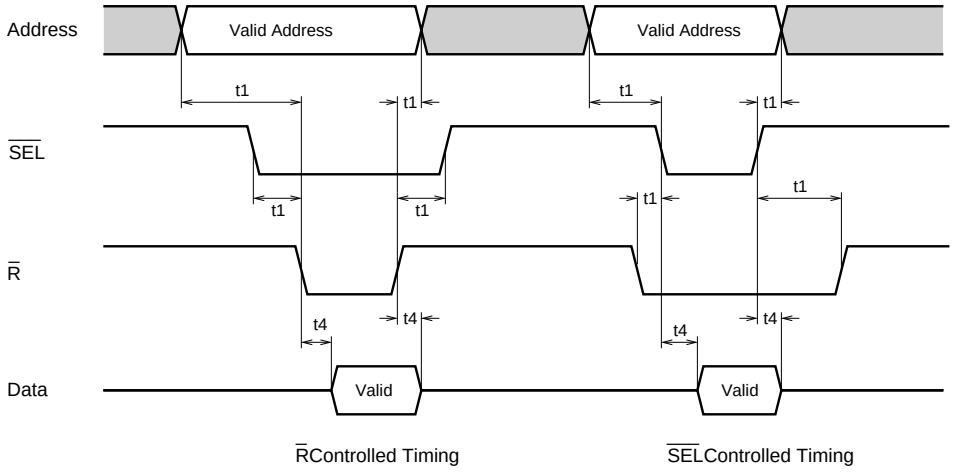

Figure 10: **Read Timing Diagram**

Timing values:

$$T_0 = \frac{1}{Controller\text{'s Clock Frequency}}$$

$$T_1 \geq 0$$

$$T_2 \geq 2.5 \times T_0$$

$$T_3 \geq 1.5 \times T_0$$

$$T_4 \leq T_0$$

Controller Setup

Any application making use of the controller should first set up all configuration registers. A sample initialization sequence is provided in pseudo c-code:

```
const ADDR_CONTROL1 = 0;
const ADDR_STATUS1 = 2;
const ADDR_STATUS2 = 3;
```

```
const ADDR_TRANSMIT = 4;
const ADDR_RECEIVE = 5;
const ADDR_ID_MASK = 6;
const ADDR_ID_FILTER = 7;
const ADDR_CLKDIV1 = 8;
const ADDR_CLKDIV2 = 9;
write_byte (ADDR_CONTROL1, 0xA1); /* 10100001 - enable receive interrupts,
                                     error interrupts, and configure as a
                                     slave node */
/* Set up the mask and filter to only acknowledge addresses in the range
   XX101000 to XX101011 (0x28 to 0x2B). Note that the top two bits are
   parity bits and are ignored for ID filtering. */
write_byte (ADDR_ID_MASK, 0xFC); /* ID mask - 11111100 (top two bits are
                                    ignored) */
write_byte (ADDR_ID_FILTER, 0x28); /* ID filter - 00101000 (top two bits
                                      are ignored) */
/* Set the clock divisor to 12 decimal (9600 bps with a system clock
   of 1.8432 MHz */
write_byte (ADDR_CLKDIV1, 0x00);
write_byte (ADDR_CLKDIV2, 0x0C);
/* Clear out the RDR and clear any error flags that might have been set */
read_byte (ADDR_RECEIVE);
read_byte (ADDR_STATUS2);
/* enable the interrupt subsystem or schedule a realtime task to
   monitor the RDR. */
enable_interrupts();
```

Frame Scheduling and Interrupt Handling

LIN frames are typically sent at periodic intervals. A LIN application may request information on sensors and update a display with that information. This may be accomplished with round-robin scheduling under a realtime OS, with interrupt-driven I/O, or a combination of the two. A full discussion of the possibilities will not be discussed here; for more information, including timing requirements, a recommended LIN API, and a configuration language that covers bit packing, scheduling, and event-driven I/O, refer to the LIN specification.

Testing Methodology

The bulk of the functional testing was performed in a simulator. However, a prototype has also been built and validated as described in Physical Testing, found below.

Simulation

Test cases were written as VHDL test benches. The basic testing architecture consisted of a two-node LIN network, one node acting as a master and one acting as a slave. The transceiver module discussed in Section 3.9 served as a digital model for a real transceiver and emulated the two-valued dominant-recessive LIN bus. The general form of a test involved writing to a register, waiting for an interrupt, and then acting on that interrupt, reading a received byte or checking the various status registers, as shown below.

```
write_register1(REG_TRANSMIT, x"10");
wait until (int2 = '1' and int1 = '1');
read_register1(REG_RECEIVE, data_read1);
read_register2(REG_RECEIVE, data_read2);
```

Test sets were written to test all of the error flags. Basic transmission tests verified that the two nodes could communicate successfully. Other test cases, like parity error and checksum error detection, required driving the bus directly. All test cases passed, confirming the design is internally consistent.

Physical Testing

The controller designed is normally a small part of a much larger system, and as such any testing beyond basic protocol compliance is highly application-dependent. Regardless, a physical demonstration was designed to show the CPU interface and loopback operation in a single-node LIN network. The design was targeted for the Nu Horizons CoolRunner-II development board, shown in Figure 11.

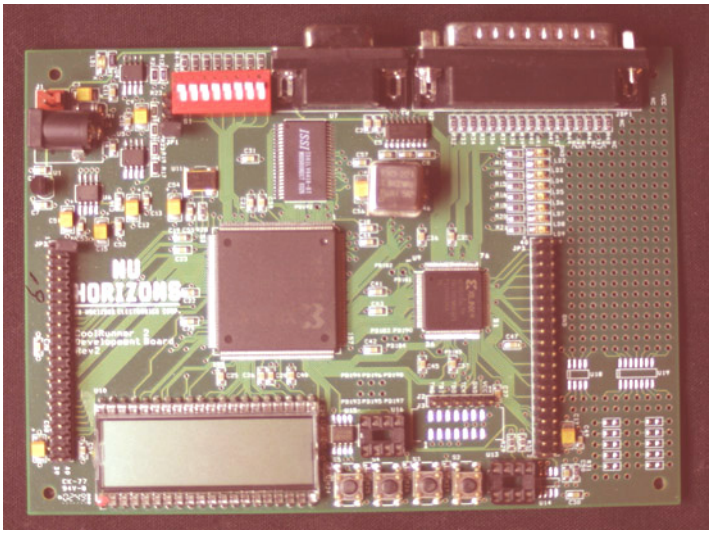

Figure 11: **Nu Horizons Development Board**

This board includes two CPLDs, one for primary development, and a smaller one for I/O interfacing. The second CPLD made this an attractive platform for prototyping, since it performed both the necessary 5V tolerant voltage translation and could also be used for simple interfacing. A simple VHDL UART provided the basic interface needed to access the LIN controller's registers through a PC serial port.

A graphical front-end was created using Labview. The interface shows the values of all of the controller's registers and simulates the operation of a simple sensor/display network as it might be found in an automobile. The values at the sensors were transmitted through the LIN network

(via loopback), subsequently received, and then used to update the displays. The functioning Labview interface is shown in Figure 12.

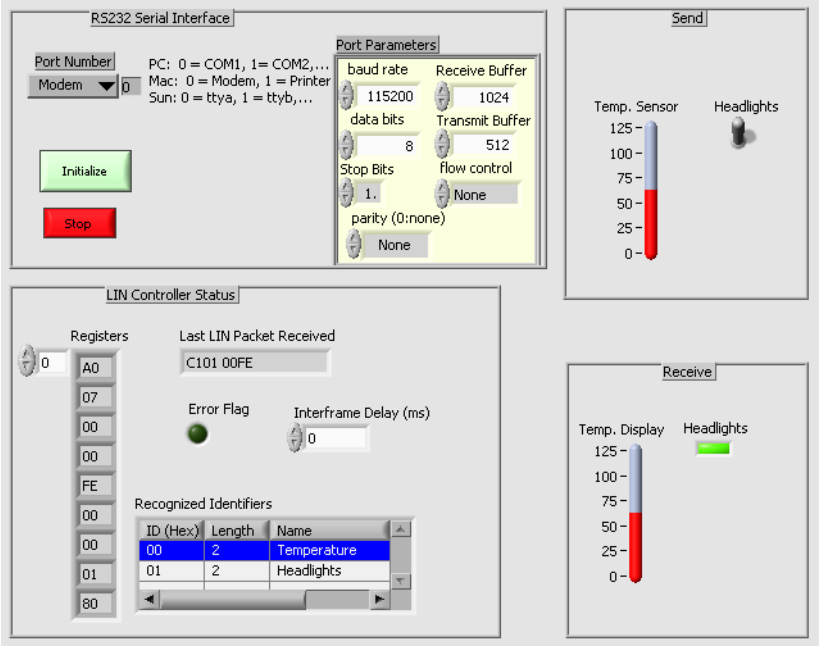

Figure 12: **Sample Labview Interface**

A loopback jumper connected between RxD and TxD emulated the LIN transceiver, which simultaneously receives the transmission currently being sent. A byte sent by the transmitting slave task would be subsequently received by the receiving slave task. Correct operation can be confirmed from Labview by comparing the sent byte to the received byte. Additionally, the error flags can be used to perform this function.

Table 6: **XC2C256 Utilization**

Macrocells	Product Terms	Registers	Pins	Function Block Inputs	Maximum Clock Rate
210/256 (82%)	698/896 (78%)	169/256 (66%)	22/173 (13%)	527/640 (82%)	25.7MHz

Conclusion

The completed controller, when synthesized for a Xilinx XC2C256 target, used approximately 80% of its resources. The maximum external clock rate, as estimated by the Xilinx synthesis tools, is 25.7MHz, more than sufficient to support the maximum LIN bitrate of 19200 bps. The timing and synthesis results are summarized in Table 6.

Functional simulation verified that the controller is internally consistent; that is, a network composed of these controllers can communicate with each other and each node correctly raises error conditions.

Physical testing verified that the implementation worked correctly in a single-node LIN network, and also verified that the configuration and status registers and CPU interface functioned correctly.

Further testing is necessary to confirm that this implementation conforms to all requirements of the LIN specification, correctly functions in a multinode LIN network, and successfully interoperates with other LIN implementations.

References

CAN in Automation. CAN Specification 2.0, Part A

CAN in Automation. CAN Specification 2.0, Part B

LIN Consortium. LIN Specification Package

CoolRunner-II Evaluation Board

Design Files

http://www.xilinx.com/products/silicon_solutions/cplds/resources/coolvhdlq.htm

Additional Information

CoolRunner-II Datasheets and Application Notes

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/15/04	1.0	Initial Xilinx release.
04/03/07	1.1	Added link to design files.