MC68HC908QY4
LIN Backlit Keypad Slave
Designer Reference Manual

M68HC08
Microcontrollers

DRM058/D
Rev. 1
07/2007

freescale.com

freescale™
semiconductor

MC68HC908QY4 LIN Backlit Keypad Slave Designer Reference Manual

By: Matt Ruff
8/16 Bit Systems Engineering
Austin, Texas

DRM058/D
Rev. 1
07/2004

freescale™
semiconductor

Revision History

Rev.	Section	Description of Change
0	Throughout	Initial release
1	Section 4; throughout	Changed headings in Table 4-1; changed to Freescale format

Table of Contents

Chapter 1. System Overview
1.1 Introduction 7
1.2 Features 8
1.3 Basic System Operation 13

Chapter 2. LIN Messaging
2.1 LIN Message Frames 15
2.2 Scheduling Tables 17

Chapter 3. Backlit Keypad Hardware
3.1 MC68HC908QY4 Backlit LIN Slave Keypad Unit 19
3.1.1 Button Interfacing 22
3.1.2 LED Backlighting 23
3.1.3 Connectors and Harnesses 24

Chapter 4. Keypad Slave Software
4.1 Overview 27
4.2 CodeWarrior, Project and File Structure 27
4.3 Application Code 29
4.3.1 Application Software State Machine 29
4.3.2 State Description 30
4.3.3 Application Data Storage 31
4.3.4 Choosing ADC Trip Points for Buttons 33
4.3.5 Software Switch Debouncing 36
4.4 LIN Software Drivers 36
4.4.1 Changes from LIN Drivers in AN2599/D 36
4.4.2 Configuring Messaging 37
4.5 Performance Issues 38
4.5.1 LIN Driver ISR vs. TIMCH0 Flag Polling 38

Chapter 5. LIN Master Module Emulation Using the LIN Spector
5.1 Overview 43

Chapter 6. Design Enhancements and Upgrades
6.1 Overview 45
6.2 Software Performance Improvements 45
6.3 Software Functionality Upgrades 46
6.4 Hardware Improvements 46
6.5 Hardware Functional Upgrades 47

Chapter 7. References and Acknowledgements
7.1 References 49
7.2 Acknowledgement 49
Appendix A. LIN_QY_Backlight_keypad_messaging_strategy_1_0.ldf 51
Appendix B. LED_Sweep.LEC 57
Appendix C. Schematic and BOM 59

Chapter 1. System Overview

1.1 Introduction

This manual describes a reference design for a local interconnect network (LIN) enabled steering wheel keypad module with light emitting diode (LED) backlighting. The design is based on the MC68HC908QY4 microcontroller and MC33689 LIN system basis chip (SBC). By placing the keypad on a LIN network, the designer is able to reduce the wiring required through the expensive clockspring wiring connection in the steering wheel and add any number of new features to the steering wheel without having to change this expensive connector. This design shows one way that a basic keypad unit can be placed onto the LIN network with simple components. The design also serves as a base design which can be added to or increased in complexity without additional connections through the clockspring.

The entire system consists of the keypad module (which is a slave on a LIN network), a satellite board (which contains more buttons and backlighting LEDs for the right side of the steering wheel), and a LIN master node. For this reference design, the LIN master node uses standard LIN tools to control and monitor the keypad unit. Design of the master module and satellite keypad is beyond the scope of this design, but the messaging used by the master to monitor switches and control backlighting levels is included.

1.2 Features

This reference design features:

- Monitoring of 13 buttons for cruise control, HVAC (heating, ventilation, and air conditioning), and radio controls on an automotive steering wheel
- LIN connectivity using LIN 1.3 standard messaging
- Autobauding to any LIN bus speed from 1 kbps to 20 kbps
- LED backlighting with 16 network-controlled brightness levels using PWM (pulse-width modulation)
- 3-wire connection to clockspring— independent of button count (LIN, V_{bat}, ground)
- LIN communications and PWM backlighting achieved using single timer module
- Based on low-cost 16-pin MC68HC908QY4 microcontroller

LIN is a low-cost serial data bus standard, based on universal asynchronous receiver transmitter (UART) hardware. LIN is targeted at low cost, low data rate networks, enabling the connection of motors, sensors, and actuators. Vehicle HVAC systems and electric power seats are good examples of systems in which LIN is used. These example systems contain multiple motors, sensors, and control panels which can easily be controlled through a relatively low-speed network (lower than 20 kbps) without impeding system performance. Networking these components rather than the traditional point-to-point wiring eliminates a large amount of wiring, reduces potential points of failure at wiring connections, and allows for the use of more advanced diagnostics.

A LIN network in the steering column might include slave nodes for a switch panel (keypad) on the steering wheel, a switch monitor in the turn signal stalk, and/or an ignition switch node. The wiring reduction in the steering column can be substantial, because many switches currently require two wires per switch to run from the switch through the steering column to a central control box. Some multiplexing of switches can be done using resistor divider networks, but this only works for a limited number of switches before requiring the use of more precise and expensive resistors.

Figure 1-1 shows a typical steering column wiring with a few extra features. In this example, at least 17 wires are required to enable these features in the steering column. Depending upon the complexity of functions and switches in the turn signal stalk, the number of wires could easily be larger.

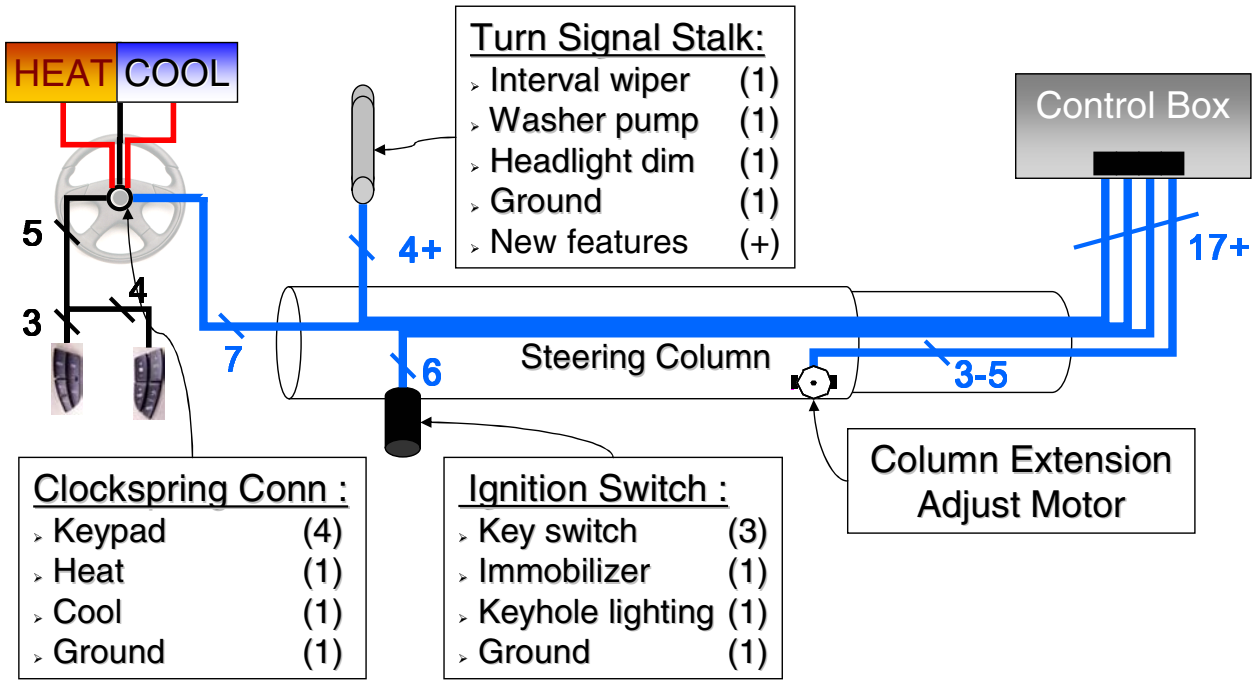


Figure 1-1. Typical (Non-LIN) Steering Column Wiring Diagram

Figure 1-2 shows how using a LIN network in the steering column can eliminate 14 or more wires in this steering column, which can reduce the size and conductor count of the connectors at every node. Additionally, connectors at each node can now be made virtually identical, decreasing cost through volume purchasing and increased manufacturability.

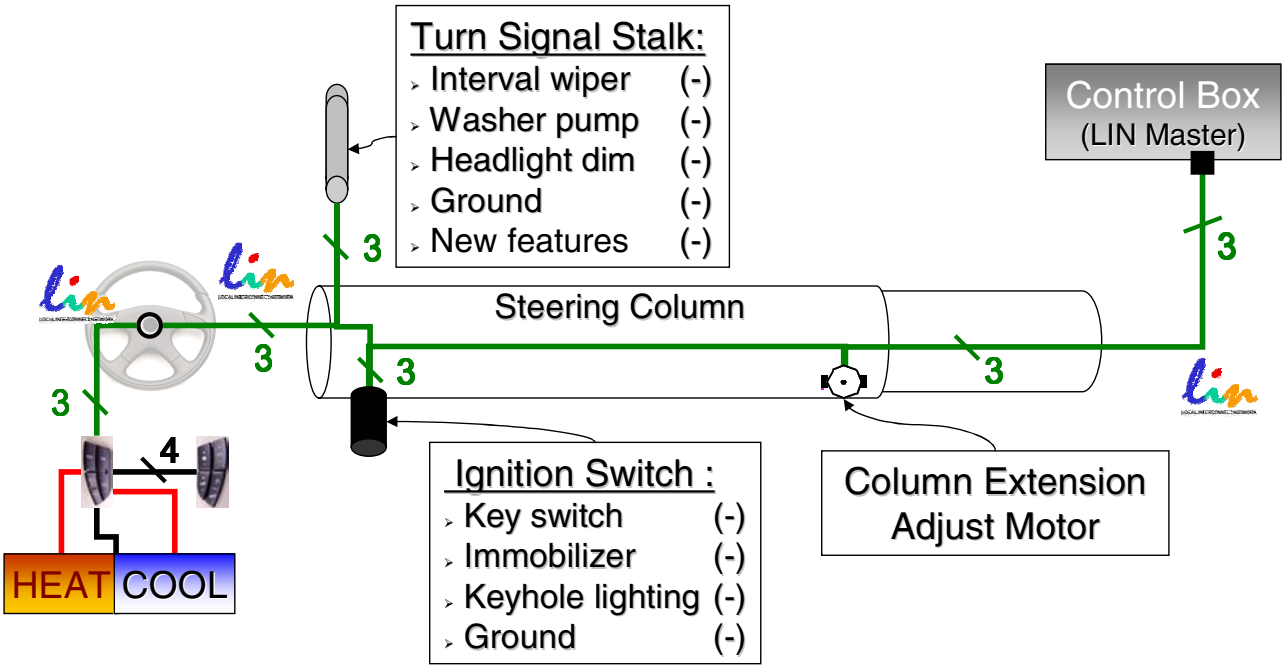


Figure 1-2. Steering Column LIN Network

In the clockspring mechanism, conductor count and connector standardization are important concerns. The clockspring links the wiring in the column to the devices in the steering wheel, which must rotate through a large range of motion and maintain electrical contact. The wheel might contain such diverse applications as keypads, airbag, heating and cooling circuitry, and lighting. Clocksprings are relatively expensive components and increase in cost with increasing numbers of conductors. In a traditional system, such as shown in Figure 1-1, as many as seven conductors are required to support the keypad, heating, and cooling functions alone. To add these functions to a traditional system that was not originally designed with them, the manufacturer must absorb the cost of the bigger clockspring, increase the cost of the assembly, or stock multiple clocksprings to accommodate all levels of vehicle options. In a LIN system, however, only three conductors and a single clockspring with a standard single connector are required for any level of functions.

Conversion to LIN not only allows for cost reduction while maintaining the same level of functionality, it allows for dramatically increased design flexibility in the steering wheel applications without adding any new wiring. If the designer is able to add features without adding wires through the clockspring, features such as adaptive backlighting, driver grip pressure sensing, or even fingerprint recognition can be added or removed from a vehicle for only the cost of the application itself. Some of these features are discussed in Chapter 6. Design Enhancements and Upgrades.

Figure 1-3 shows a basic overview of a LIN enabled steering wheel keypad and how it connects to the master node of the system.

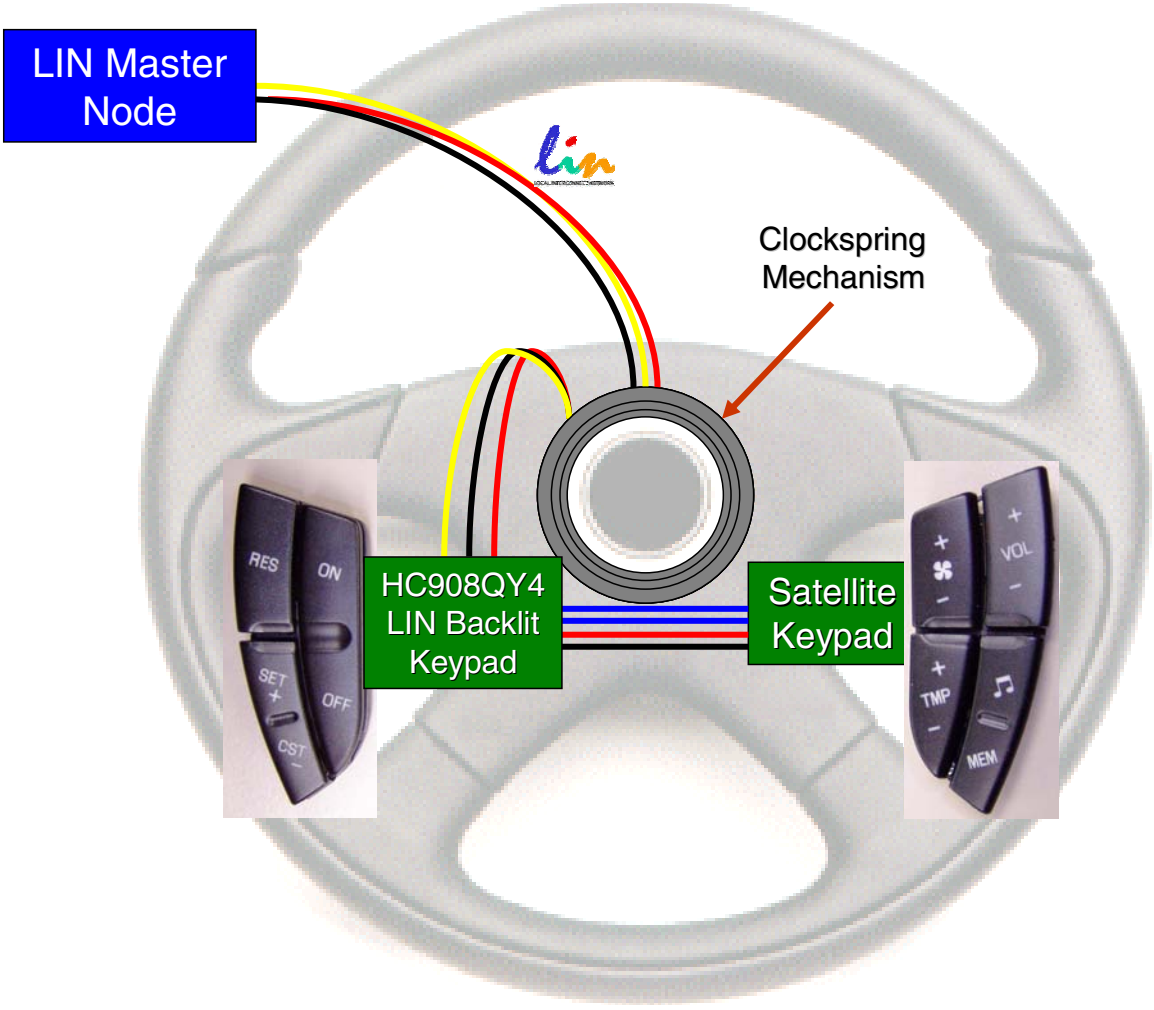


Figure 1-3. Reference Design System Overview

Much more elaborate features can be implemented, but the two basic functions are monitoring a set of switches and illuminating those switches with LEDs. Some additional features and enhancements to this basic design are described in Chapter 6. Design Enhancements and Upgrades.

Figure 1-4 is a block diagram of the keypad that shows the basic components and interconnections. One of the other key features of this design is that it uses the very small and inexpensive MC68HC908QY4 microcontroller unit (MCU) to operate the LIN communications, PWM backlighting, and switch monitoring. This is accomplished with only a single two-channel timer module and general-purpose input and output pins. Much of the other circuitry required in such a design, such as voltage regulation from automotive battery voltage (V_{bat}) down to 5 V, LIN physical interfacing, and V_{bat} level LED drive are accomplished with the use of the MC33689 LIN system basis chip (SBC). The LIN SBC also extends the input and output capabilities of the 16-pin package of the MCU to allow the addition of more switches or output devices.

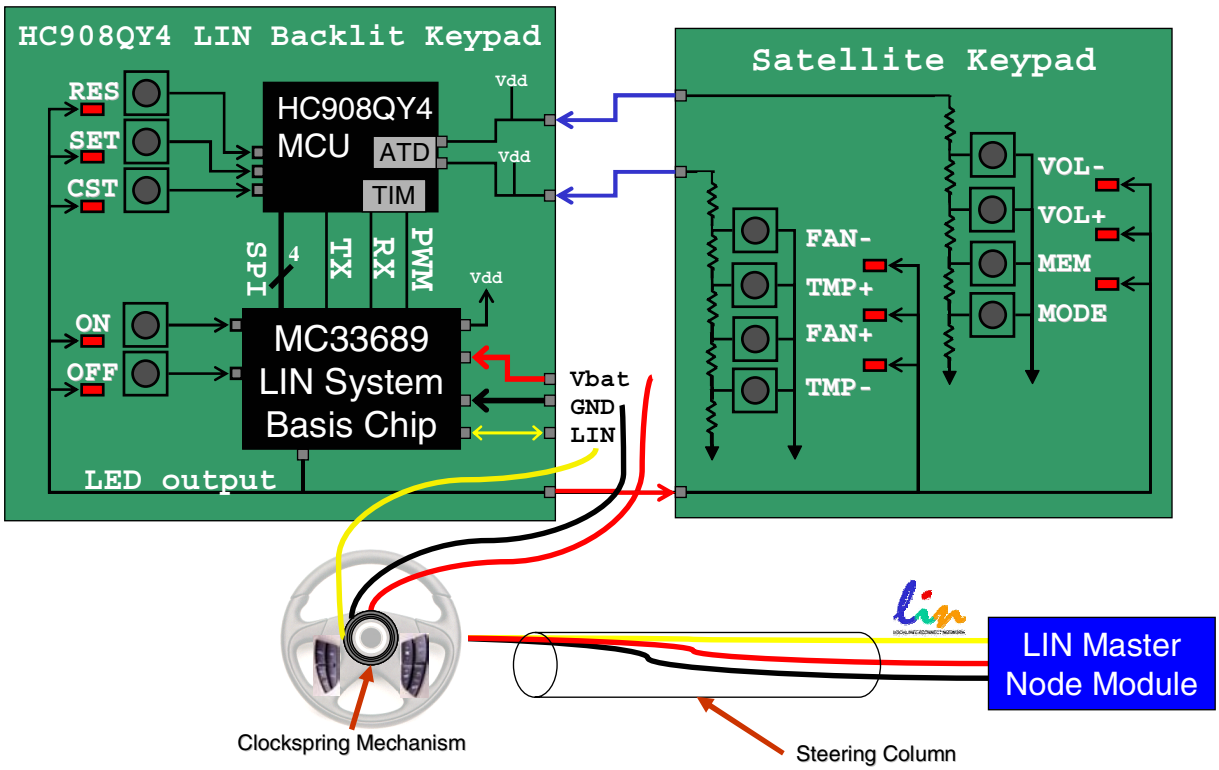


Figure 1-4. Keypad Block Diagram

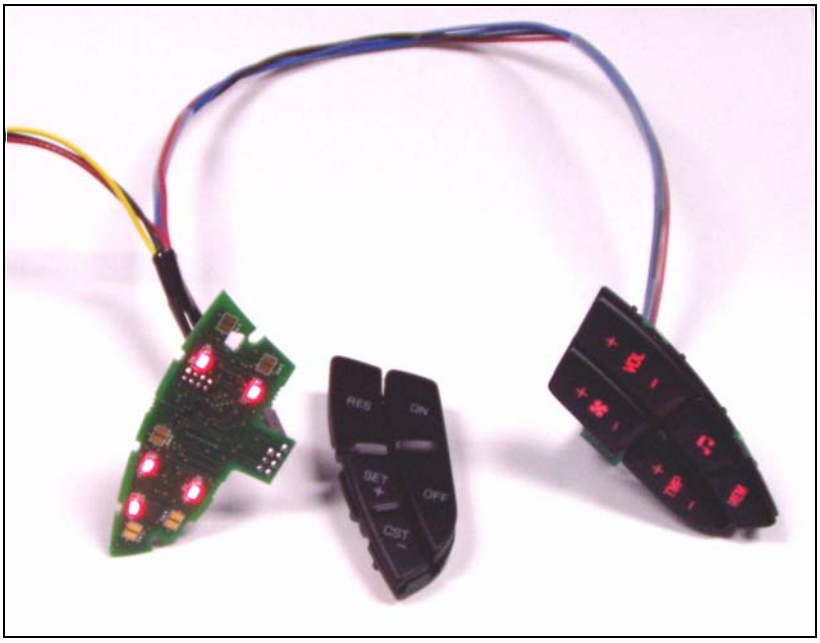


Figure 1-5. MC68HC908QY4 Backlit LIN Keypad with Satellite Keypad and Harness

1.3 Basic System Operation

The core components of the system are the LIN backlit keypad module and the LIN master module. This reference design focuses primarily on the design of the keypad unit, which is a slave on the LIN network. The functions of the master node are emulated using a standard LIN development tool called the LIN Spector™ from Volcano Automotive Group and are described in more detail in Chapter 5. LIN Master Module Emulation Using the LIN Spector.

In a LIN network, the master node initiates all communications on the network. In this system, it simply polls the keypad for button status using request messages and transmits LED lighting levels via command messages. The keypad constantly checks the switches for status, updates the PWM output for the LEDs based on the current backlighting level, and responds to any recognized LIN message header that arrives. LIN messaging details are explained in Chapter 2. LIN Messaging.

LIN Spector™ is a trademark of Volcano Automotive Group.

Chapter 2. LIN Messaging

The keypad is designed to respond to four LIN messages; three are for status and control and one is the LIN standard sleep message. The three specific status and control messages can be modified easily in `LINmsg.c`; however, adding messages may affect code performance. Refer to 4.5 Performance Issues for more detail.

2.1 LIN Message Frames

The following LIN message frames are used:

Table 2-1. LIN Message Frames

LIN ID (with parity)	Frame Name	Frame Description	Byte 0	Byte 1
0xCA	`LIN_Stat`	J2602 and LIN 2.0 compliant status byte	Application status, LIN communication status	—
0x8B	`KEY_STATUS`	Keypad button press status	Cruise control button status	HVAC and radio button status
0x4C	`INT_LIGHT_CMD`	Interior lighting command message	Backlighting ON/OFF and level data	—
0x80	—	LIN 1.0 system sleep message	—	—
0x3C	—	LIN 1.3+ system sleep message (not currently supported)	0x00	—

Full details of the messaging can be found in the LIN description file (LDF) found in the reference design software download file and in Appendix A. LIN_QY_Backlight_keypad_messaging_strategy_1_0.ldf. The format for this file is part of the LIN standard, which means that the LDF file describes the LIN messaging in a standard way so that it will work with all LIN compliant tools.

The LIN_Stat message is not used in this reference design, but it is included to show how this LIN and SAEJ2602 standard status byte might be used in a system. The slave supplies to the master a 1-byte status update that contains LIN bus error conditions and application status information. This allows the master to rapidly poll the slave node status by requesting a 1-byte message without having to ask for detailed status data, which requires longer messages.

The LIN driver in this reference design does track bit errors and checksum errors, however they are not currently reported in the LIN_Stat message. The application status information in bits 0 through 4 are defined by the application designer. The only reserved value is 0x00, which indicates that there is no data to report. The application status field is updated in the reference design (as an example) whenever a button is pressed. The encoding of the LIN_Stat message byte allows the master node to rapidly request this 1-byte status update and monitor the most significant bit (MSB) for LIN bus error conditions and the least significant bit (LSB) for application state changes.

The KEY_STATUS and INT_LIGHT_CMD messages are the primary messages used in this reference design. Because only 13 bits are required to report button status (with one additional bit that reports the pressing of any button), only a 2-byte status response is required in message KEY_STATUS. The added latency (2 bytes instead of 1 byte) is minimal. Therefore in this reference design, the master simply uses this message to monitor the condition of the keys. The KEY_STATE_CHG signal (bit 7 of byte 0) serves the same function as the LSB of the LIN_Stat message.

The INT_LIGHT_CMD is a command message the master node uses to report the interior lighting levels to the slave node. A 4-bit value represents the brightness level required in increments of 6.65% per bit. The MSB is a toggling bit that indicates whether the backlighting should be turned on or off. If this bit indicates that the lighting should be off, the brightness level is ignored and the backlighting is disabled.

The LIN drivers included with the reference design recognize the standard LIN 1.0 system sleep request message. In LIN 1.3 and later, this message is defined as part of the 0x3C “diagnostic master request frame” message set, where the first data byte is equal to zero. Currently,

the application recognizes only the older version of this command (from LIN 1.0) which was simply defined by the reception of identifier 0x80. When this message is received, a bit flag called `LINsleep` is set, allowing the application to recognize this command from the master and decide how to manage network sleep conditions. The main application routine polls the flag. If the flag is set, it calls the `GoToSleep()` routine. A small code stub has been inserted into `LINdriver.c` to show how to expand the driver to also work with the 0x3C message style, if desired. The stub reads as follows:

```
if (Id == DIAGMSTRREQ)   //Defined in LINdriver.h, should be 0x3C
{
// Set a temp latch that "diagnostic Master Req Frame" ID received
// then check data[0] when rcvd, if ==0x00, then set sleep flag
}
```

2.2 Scheduling Tables

A number of scheduling tables are defined in the LDF file. A scheduling table is a defined sequence of message frames which are sent out in a particular order at particular times. Different scheduling tables are defined in the LDF file primarily to facilitate testing and debugging of the slave node. Most LIN tools are capable of emulating the master node in a LIN network if message frames and schedule tables are defined. More advanced tool features also allow switching between schedule tables.

For example:

1. Run the `FAST_POLL` table (which sends the `LIN_Stat` message ID)
2. Whenever the LSB returns a value of 1, switch to the `CHECK_SWITCHES` table
   (Requests full button status using the 2-byte `KEY_STATUS` frame)
3. Switch back to `FAST_POLL` table to poll the node more rapidly
4. Switch to `LIGHTS_ONLY` for one schedule table cycle whenever the master node senses a change in the required backlighting

Table 2-2 provides a brief description of the included schedules.

Table 2-2. Schedules Included in this Reference Design

Schedule Table Name	Function
FAST_POLL	Used to rapidly check node status
CHECK_SWITCHES	Sends only key_status request message frame to monitor key conditions
LIGHTS_ONLY	Sends only int_light_cmd command message frame to send backlighting settings
CHECK_SWITCHES_AND_LIGHT	Alternately sends KEY_STATUS request message frame to monitor key conditions and INT_LIGHT_CMD command message frame to send backlighting settings

Chapter 3. Backlit Keypad Hardware

3.1 MC68HC908QY4 Backlit LIN Slave Keypad Unit

The hardware design for the MC68HC908QY4 backlit LIN slave keypad happens to be based upon a keypad for the 2003 model Ford Expedition™ Eddie Bauer™ Edition (Ford Replacement Part#: 1L2Z 9C888 BA, Description: NL SW AS), but any similar keypad could be used. The reference design hardware replaces the left switch unit and is designed to allow the button assembly from the factory unit to be removed and placed upon the reference design board. The right switch unit (satellite keypad) is intended to be used as is, without modification, except perhaps changing the LEDs and current-limiting resistors as appropriate to match the left switch unit color and intensity.

To remove the existing PCB from the Ford keypad, two T9 Torx® or star-drive screws must be removed. These can be found between and behind the two columns of buttons at the top and bottom of the switch unit. The Torx driver must be wedged between the columns of buttons to reach the screws.

Torx® is a registered trademark of Camcar Div. of Textron Inc.

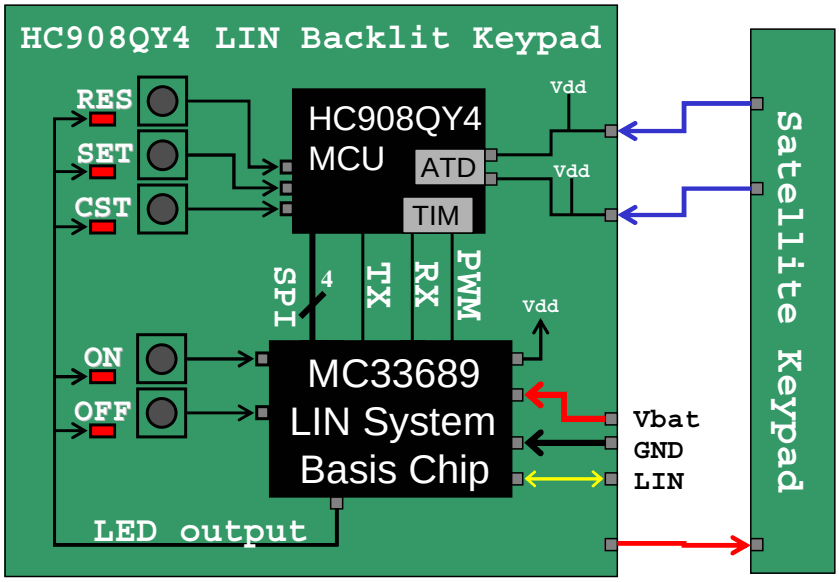


Figure 3-1. HC908QY4 Backlit LIN Slave Keypad Unit Block Diagram

The MC68HC908QY4 MCU has only 16 pins, including V_{DD} and ground, and it must monitor 13 switches, communicate over the LIN bus, and operate LED backlighting through a PWM output. This is accomplished by combining the MCU with a LIN SBC. Four pins are required to connect the SBC to the MCU, but the SBC combines many functions of voltage regulation, LIN physical interfacing, additional input and output circuitry, and full diagnostics and circuit protection, which compensates for using the four MCU pins.

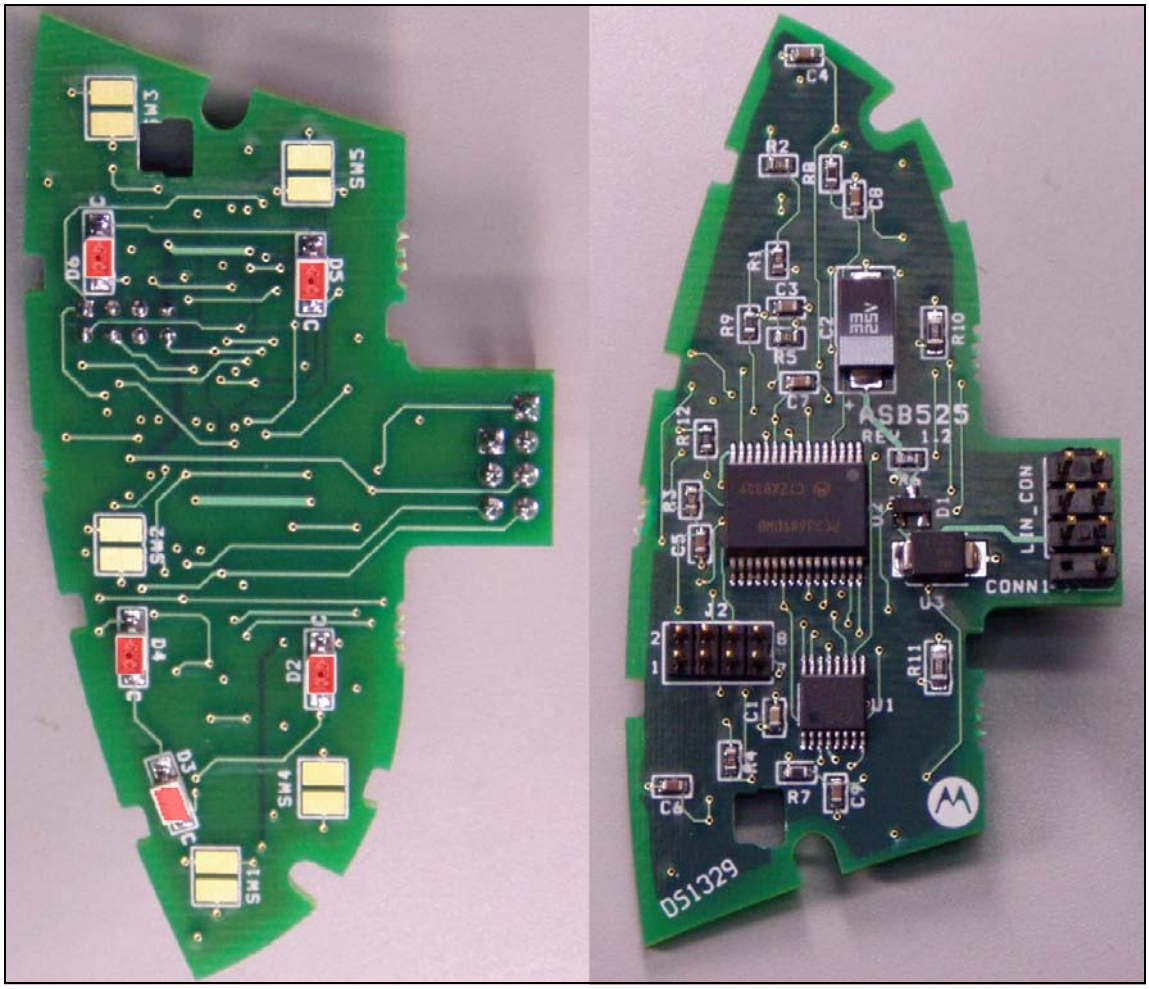


Figure 3-2. HC908QY4 LIN Backlit Keypad Unit — Switch Side (L) and Component Side (R)

Full schematic of this board can be found in the download files for this reference design and in Appendix C. Schematic and BOM.

3.1.1 Button Interfacing

The primary function of a steering wheel keypad is to monitor the status of buttons to provide an input device for the vehicle driver to easily control HVAC, radio, cruise control, or other vehicle systems. The chosen base design requires 13 buttons to be monitored, which is accomplished through three different methods. Buttons may be:

- Directly input to the MCU and read as digital inputs
- Connected to the MCU through analog-to-digital (ADC) inputs and decoded from analog voltages
- Connected directly to inputs on the SBC

The three cruise control runtime functions most used by the driver are:

- Setting a target speed (SET)
- Resuming a pre-set speed after a temporary disengagement of the cruise control (RES)
- Coasting to allow the vehicle to slow down gradually resetting the target speed (CST)

The buttons controlling these three functions are set up as standard 5-V inputs directly connected to the MCU input pins with external pullup resistors to V_{DD}.

The two buttons that enable and disable the cruise control function (ON and OFF) are connected to the SBC wake-up switch inputs with vehicle battery voltage (V_{bat}) level pullup resistors. This allows the buttons to be read without using additional pins on the MCU, and the status of these button inputs can be read through the SPI interface to the SBC.

Another set of buttons is located on the satellite keypad board, which is located on the right side of the steering wheel. The buttons on this assembly are arranged in two groups of four buttons each, connected to resistor ladders which yield a different voltage depending on which button is pressed. Figure 3-3 shows the block diagram of this assembly.

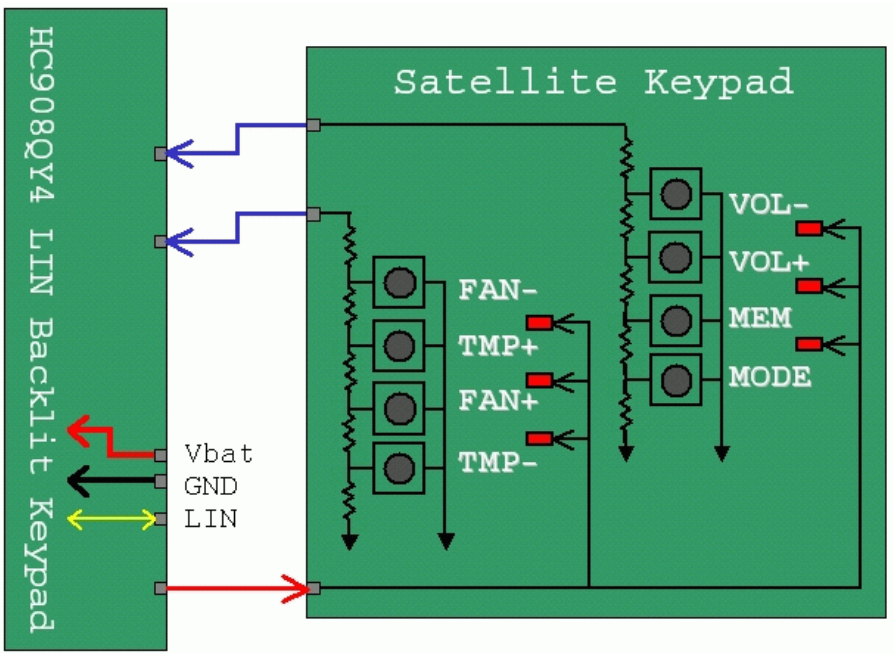


Figure 3-3. Satellite Keypad Block Diagram

Each of these two groups of buttons is connected through short wires within the steering wheel to ADC inputs on the MCU. In this way, the MC68HC908QY4 keypad unit can determine which buttons are being pressed on the right side keypad. Some combinations of multiple key-presses are possible and some might yield a different voltage; it is the function of the software to determine the correct interpretation of this combination. Details of this are discussed in 4.3 Application Code.

3.1.2 LED Backlighting

The SBC has three high-side protected drivers, allowing it to power large loads such as lamps and relays. Two of these drivers (HS1 and HS2) can further be controlled through a PWM input pin, allowing a PWM signal to be ANDed with the switch setting. The result is the ability to PWM larger loads and loads which must be switched at V_{bat} levels. For this reference design, all LED backlighting is connected to HS1 and controlled by this driver and the PWM signal is generated on TCH0 pin of the MCU's timer. Further enhancements to the backlighting can be

accomplished through alternative circuitry and are discussed in Chapter 6. Design Enhancements and Upgrades.

3.1.3 Connectors and Harnesses

Both keypad units are separate assemblies which are mounted to opposite sides of the steering wheel, requiring a wire harness to connect them to each other and to connect to the LIN bus through the clockspring mechanism. Figure 3-4 shows the connections required to create the wiring harness for the keypad assembly. The connector used for the clockspring connector is a standard Freescale Semiconductor LIN development tools connector, which allows this assembly to easily interface any of Freescale's LIN master evaluation boards or LIN kit boards.

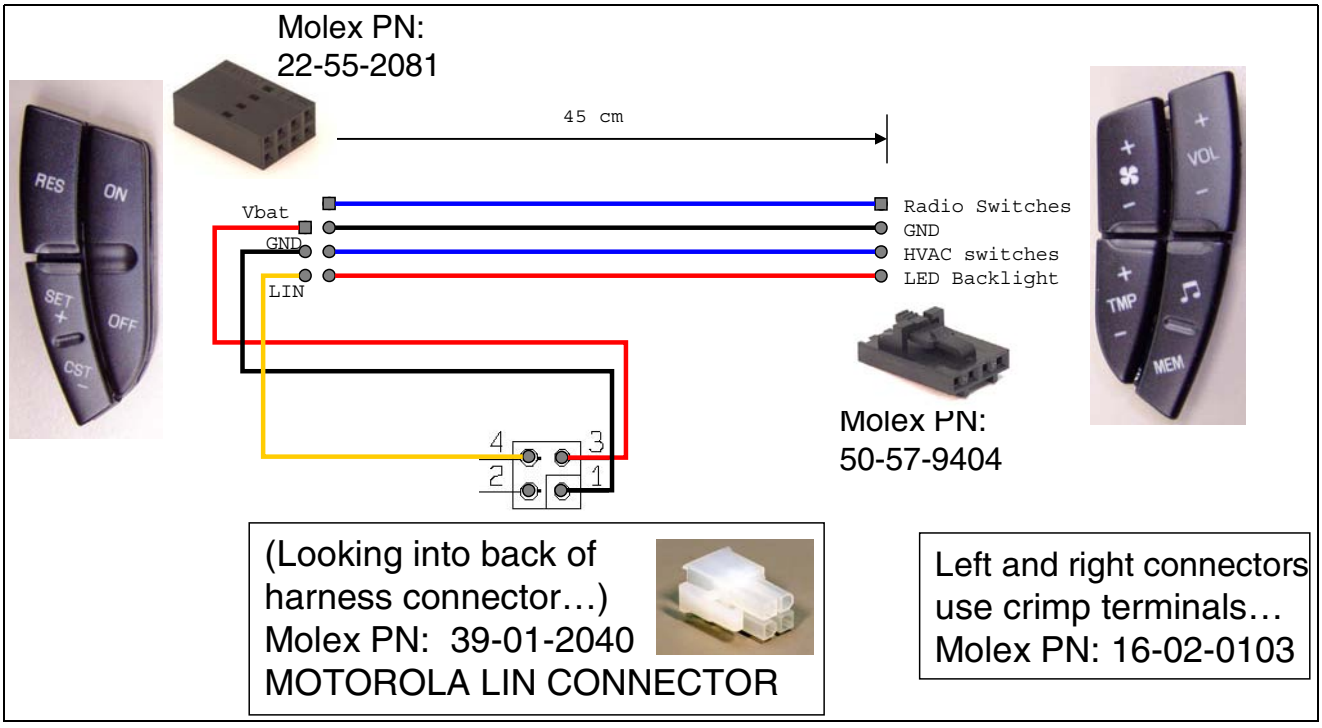


Figure 3-4. Steering Wheel Keypad Wiring Harness Diagram

The last major hardware interface in the design is the debugging and programming interface. This connector would not be populated in production designs, and likely would not be designed into the final hardware layout, but it is provided here as a reasonable compromise for development purposes. It allows the user to reprogram the MCU, but debugging with this interface is extremely difficult due to the sharing of the PTA0 pin between the PWM backlighting function and the monitor mode debugging interface.

CAUTION: *Do not plug the connector in backwards. Doing so can cause damage to components on the board. Improvements to this debugging connector interface are discussed in Chapter 6. Design Enhancements and Upgrades.*

Chapter 4. Keypad Slave Software

4.1 Overview

The software for the LIN keypad is fairly straightforward and covers only the basic required functions of the keypad application. This design is engineered to get a high level of performance from a relatively small and inexpensive microcontroller. As a result, there are some design challenges that are easy to overlook in the code. Most of the code is written in C, but several key routines and code sequences are written in assembly code to better optimize performance.

4.2 CodeWarrior® Project and File Structure

The CodeWarrior 3.0 project file (`QY4_LIN_Keypad.mcp`), located in the sample directory, is included with this reference design and contains all the code required to build the application. The code was originally based on the MC68HC908QY4 LIN drivers detailed in AN2599/D: *Generic LIN Driver for MC68HC908QY4*. Details on those drivers and how they can be used can be found in that application note. See Chapter 7. References and Acknowledgements.

CodeWarrior® is a registered trademark of Metrowerks, Inc., a wholly owned subsidiary of Motorola, Inc.

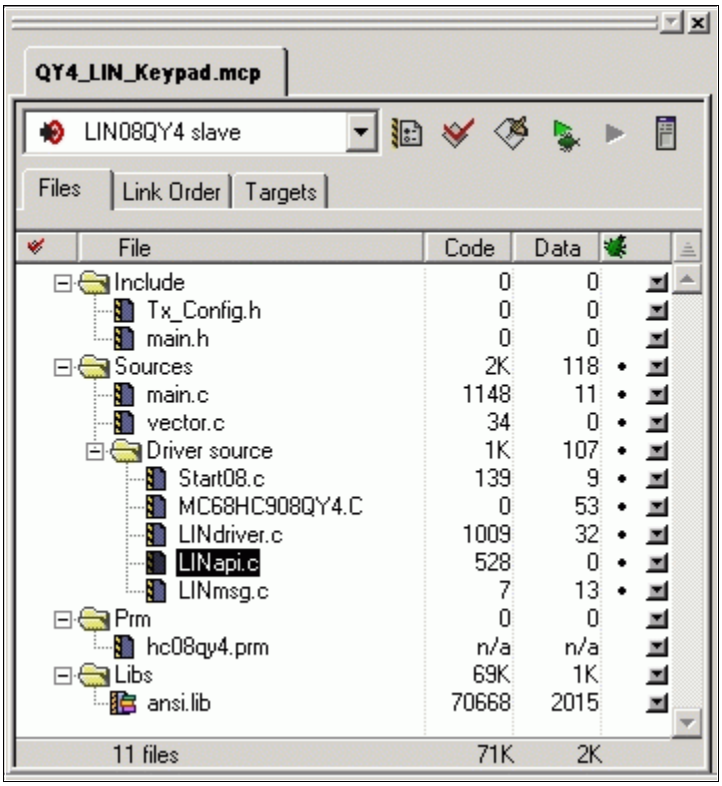


Figure 4-1. CodeWarrior 3.0 HC08 Project (`QY4_LIN_Keypad.mcp`)

The application code is contained in `main.c` with application-specific variable declarations, macro definitions, and data structure definitions in `main.h`.

The LIN driver consists of five files:

- **`LINdriver.c`** — contains the main driver code
- **`LINdriver.h`** — header file for the driver
- **`LINapi.c`** — contains all the driver API functions (only `LIN_Init` routine is used)
- **`LINmsg.c`** — where all LIN message frames are defined
- **`Tx_Config.h`** — header file containing the transmission pin definitions

The driver also requires these standard files and libraries:

- **`Start08.c`** — standard start-up routines
- **`MC68HC908QY4.h`** — header file for the MC68HC908QY4 MCU
- **`MC68HC908QY4.c`** — C file for the MC68HC908QY4 MCU with data structure instantiations
- **`ansi.lib`** — ANSI library file
- **`vector.c`** — interrupt vector definitions for MCU
- **`hc08qy4.prm`** — memory locations (ROM and RAM) in the MCU

4.3 Application Code

The application code for the reference design is in `main.c` and deals with monitoring the status of the buttons, maintaining the data in the LIN message buffers with this data, and updating the LED backlighting PWM signal.

4.3.1 Application Software State Machine

The state diagram for the main application is shown in Figure 4-2. Because the states are simple, they are not explicitly written into the application code and are shown for clarity. The state machine is event driven where all state transitions are gated by the events shown in the figure rather than by time triggering.

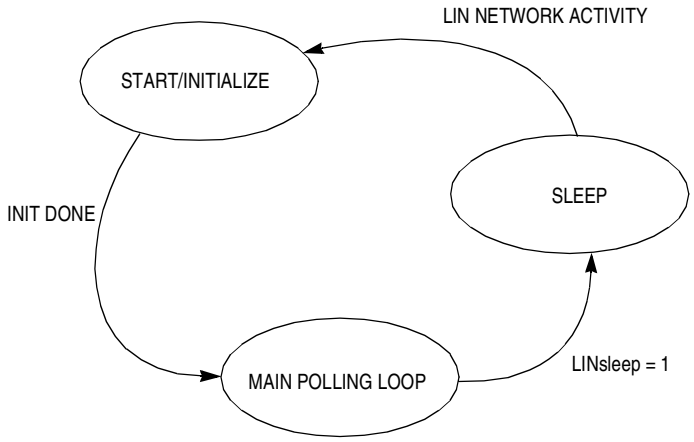


Figure 4-2. HC908QY4 LIN Backlit Keypad Software State Diagram

4.3.2 State Description

The following is a detailed description of the states:

- START/INITIALIZE — After reset, the software initializes the oscillator trim value to 0x00 to get the fastest possible bus clock (3.2 to 4 MHz), sets the LIN driver to the unsynchronized state, sets up the timer and ADC, sets up the input and output settings for the general-purpose pins, and initializes the SBC to get ready for LIN messaging. Additionally, the LIN message buffers are cleared.
- MAIN POLLING LOOP — After initialization is complete, the software enters the main running state. This is a polling loop that reads each of the three sets of buttons via direct inputs, SBC inputs using SPI data, and ADC measurements. This data is stored in a temporary latch which is transferred once to the LIN message buffers at the end of the main loop. The polling loop is gated by the period of the timer module. When the timer reaches a certain period, the scheduling flag is set, which triggers one poll of the buttons. Throughout the polling loop and every time that any delay might be encountered, the timer channel 0 flag is polled to determine whether the PWM signal requires updating. The PWM must be managed in software rather than hardware due to the sharing of the timer module and the time-critical nature of LIN bus

traffic. The LIN driver state machine is allowed to run while the application is in this state because this is the only time interrupts are enabled. The LIN drivers are discussed in more detail in section 4.4 LIN Software Drivers.

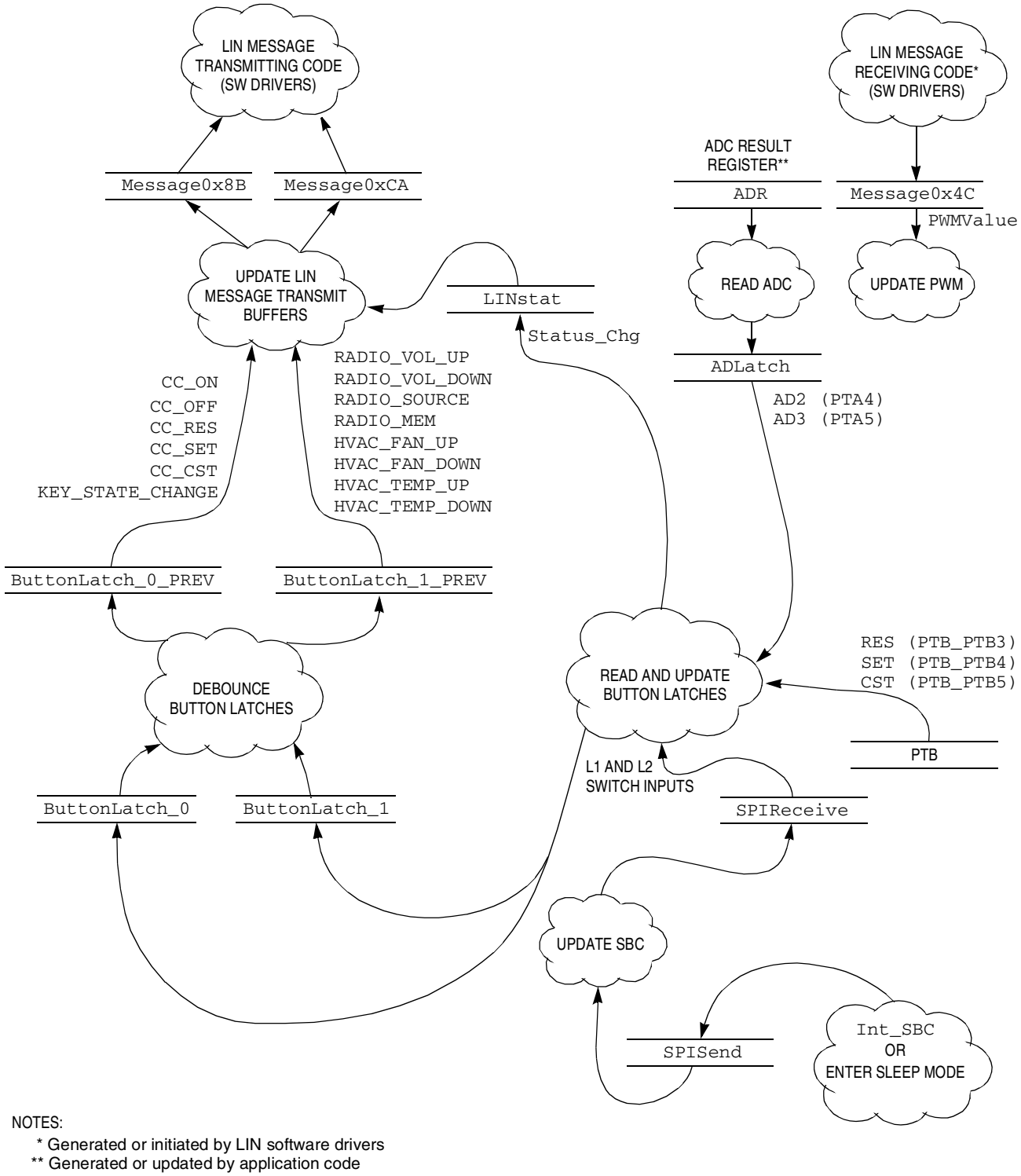
- SLEEP – After successful reception and interpretation of the global LIN sleep command message, the software enters the SLEEP state. This state serves as a trap for the sleep command. In the current design, an infinite loop is entered, but in a final design, this state would place the SBC into its sleep mode, which powers off the MCU. The state is exited only when there is message traffic on the LIN bus which triggers the SBC to re-enable the regulator which causes a power-on reset (POR) in the MCU. The current software also recognizes only the older, LIN 1.0 form of the LIN sleep command (ID=0x80).

4.3.3 Application Data Storage

A description of the key data storage flow is shown in Figure 4-3. This diagram illustrates the interaction of data between the different software processes, hardware components, and system events. This diagram should be created before software is written and serves as the basis for creating all data storage structures required in the application code itself.

The bubbles indicate processes that manipulate or generate data in the system. Data storage locations are indicated by text with horizontal lines above and below and use the exact names of the data storage structures used in the code itself (e.g., `ButtonLatch_1`). In some cases, these storage areas are simply registers in the MCU memory map. In other cases, they are the LIN message buffers created in RAM for sending and receiving LIN messages.

Data flow is shown as arrows with accompanying text to explain what data is being moved. All text shown in `COURIER` font corresponds to a variable or constant name used in the C code.



NOTES:
* Generated or initiated by LIN software drivers
** Generated or updated by application code

Figure 4-3. HC908QY4 LIN Backlit Keypad Software Data Flow Diagram

4.3.4 Choosing ADC Trip Points for Buttons

To choose the ideal cutoff values for the ADC measurements of the satellite keypad, calculate the net resistance of both channels for two sample units. This will provide reference resistance levels, which can be used to extrapolate ADC measurements for the other points. Then average the two sets of extrapolated numbers and calculate the mean value of the two resistance values desired. This final mean value will typically be the ideal ADC trip point value.

For example, the cutoff value for the RADIO_MEM button was calculated as follows:

1. Average estimated ADC reading with RADIO_MEM pressed:
   `(78.8+78.6)/2 =  78.7`
2. Average estimated ADC reading with RADIO_SOURCE pressed:
   `(121.2+120.9)/2 = 121.05`
3. Mean ADC Value:
   `(121.05+78.7)/2 = 99.875`
4. HEX equivalent:
   `0x63`

Table 4-1 shows the complete set of resistance measurements for both sample units, all estimated ADC readings (in decimal and hexadecimal), and the calculated ADC cutoff values. The net resistance measurements for the satellite keypad units were measured between pins 1–2 and pins 2–3 (radio and HVAC data respectively) of the satellite keypad connector. All calculations are based on these measurements and a 1-kΩ resistor to V_{DD} located on the HC908QY4 LIN keypad module.

Table 4-1. Satellite Keypad Resistance Measurements and ADC Cutoff Calculation Data

Button Pressed	Net Resistance Measured		Est. Voltage Reading		Est. ADC Value (DEC)		Est. ADC Value (HEX)		CALC ADC Cutoff	
	Unit 1	Unit 2	Unit 1	Unit 2	Unit 1	Unit 2	Unit 1	Unit 2	DEC	HEX
RADIO_VOL_DOWN	74.10	73.40	0.34	0.34	17.7	17.5	11	11	30.7	1E
RADIO_VOL_UP	207.30	206.50	0.86	0.86	44.0	43.8	2B	2B	61.3	3D
RADIO_MEM	445.00	443.00	1.54	1.53	78.8	78.6	4E	4E	99.9	63
RADIO_SOURCE	899.00	895.00	2.37	2.36	121.2	120.9	79	78	166.6	A6
<none>	4890.00	4800.00	4.15	4.14	212.5	211.9	D4	D3		
HVAC_FAN_DOWN	130.20	130.40	0.58	0.58	29.5	29.5	1D	1D	48.4	30
HVAC_TEMP_UP	355.90	356.30	1.31	1.31	67.2	67.3	43	43	89.6	59
HVAC_FAN_UP	778.00	778.00	2.19	2.19	112.0	112.0	70	70	136.8	88
HVAC_TEMP_DOWN	1710.00	1709.00	3.15	3.15	161.5	161.5	A1	A1	186.1	BA
<none>	4650.00	4640.00	4.12	4.11	210.7	210.6	D2	D2		

For a clearer understanding of these cutoff frequencies, Figure 4-4 shows the extrapolated ADC readings for unit 1 and unit 2 with 10% error bars and the calculated ADC cutoff values to be used in the code. The calculated cutoff values (CALC ADC Cutoff) are exactly between the adjacent resistances, which gives maximum clearance from the adjacent values and allows for the 10% possible variance in resistor values in manufacturing.

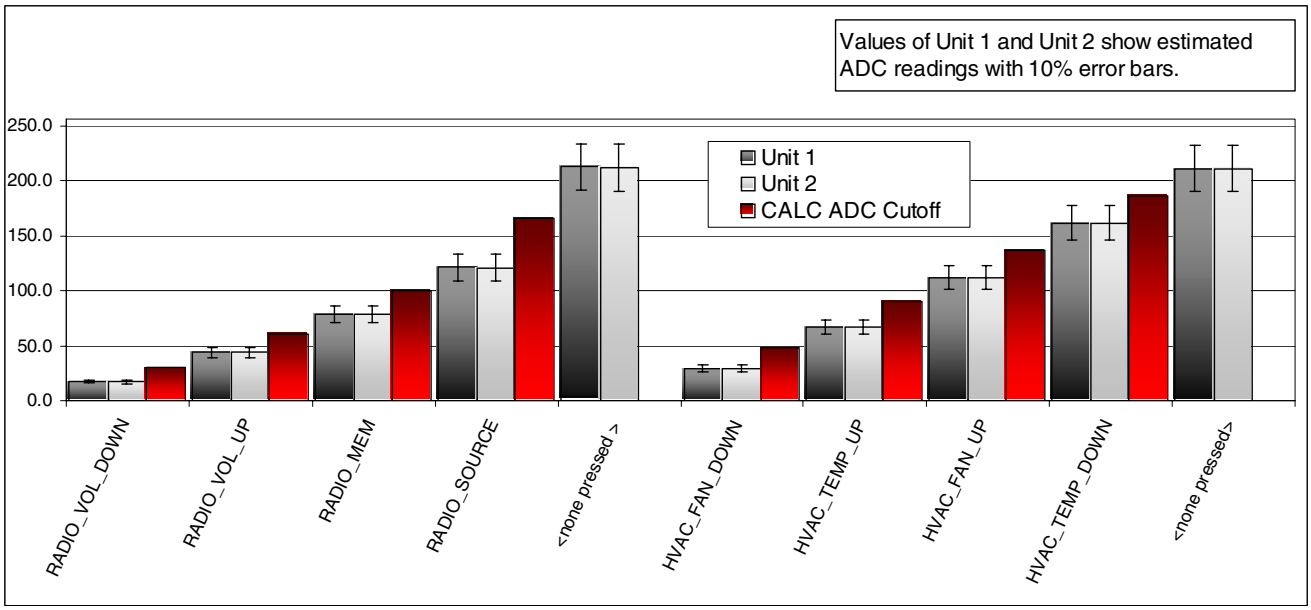


Figure 4-4. ADC Cutoff Values

These cutoff target values are then used to scan through the ADC readings in the code as follows:

```
if      (ADLatch < 0x1E) ButtonLatch_1.Bits.RADIO_VOL_DOWN   = 1;
else if (ADLatch < 0x3D) ButtonLatch_1.Bits.RADIO_VOL_UP     = 1;
else if (ADLatch < 0x63) ButtonLatch_1.Bits.RADIO_MEM        = 1;
else if (ADLatch < 0xA6) ButtonLatch_1.Bits.RADIO_SOURCE     = 1;
```

In this way, one and only one button value will be chosen on each scan, and the lowest resistance will dominate. It is possible to have some combinations of multiple button presses, but some are impossible due to mechanical linkages of the buttons. Due to the unlikely occurrence of these multiple button presses by the vehicle operator (these switches are generally operated with the thumbs only during vehicle operation), these cases are not addressed specifically in this reference design. The most likely combination is if the driver accidentally pressed `RADIO_VOL_DOWN` and `RADIO_SOURCE` simultaneously by pushing on the gap between them. This case would be interpreted as `RADIO_VOL_DOWN` while both buttons are pressed due to the disparity in resistance values, which is the safest fault condition for the system. Managing all multiple button press conditions are beyond the scope of this design, but many methods can be used in software.

4.3.5 Software Switch Debouncing

A very simple and effective software switch-debouncing mechanism will ensure that a button must be pressed for at least two cycles of the main polling loop. Because of the way the application code is written, this will require that a button be pressed for at least 4 ms before it registers as being pressed. This is accomplished by establishing two latch buffers to hold the value of button status. The current button readings are always loaded into `ButtonLatch_0` and `ButtonLatch_1`, then they are bit-wise ANDed with the values from the last cycle through the polling loop stored in `ButtonLatch_0_PREV` and `ButtonLatch_1_PREV`. The resulting debounced values are loaded into the LIN message buffers for transmission over the network, and finally the debounced latches are updated for the next pass through the polling loop.

4.4 LIN Software Drivers

The LIN communications driver used in this reference design is essentially the same as the one found in AN2599/D: *Generic LIN Driver for MC68HC908QY4* with a few changes required to speed execution time and reduce final code size. Further changes to this driver or the use of another driver, such as the one in AN2503/D: *Slave LIN Driver for the MC68HC908QT/QY Family* are detailed in Chapter 6. Design Enhancements and Upgrades. Also see Chapter 7. References and Acknowledgements.

4.4.1 Changes from LIN Drivers in AN2599/D

One of the primary changes made to the usage of the LIN drivers is the added direct access of the message buffers in memory. The message buffer data structures are declared as external data structures in `main.c` to allow direct accesses, rather than requiring the use of the `LIN_PutMsg()` and `LIN_GetMsg()` application programming interface (API) calls. Because these API calls are not used, they are not compiled; this saves code space and reduces execution performance overhead.

Another change is the elimination of updating the LIN message buffer status update code in the timer interrupt service routine (ISR) in `LINdriver.c`. This reduces the execution time of the ISR. Because the `LIN_PutMsg()` and `LIN_GetMsg()` API calls are not used to update and retrieve data in the LIN message buffers, the LIN message buffer status data is not updated properly or checked anyway. This renders the buffer status update code in the ISR unnecessary. The only API call used in the reference design software is the `LIN_Init()` routine.

Several other performance improvements were made to speed execution of the timer ISR which runs the LIN driver.

For example, the following code:

```
for (i=0 ; i <= ((MessageCountTbl[MessageIndex] & 0xF) - 1);i++)
{
        FrameBuffer[i] = *(MessagePointerTbl[MessageIndex] + i);
}
```

was replaced with:

```
cnt_limit = ((MessageCountTbl[MessageIndex] & 0xF) - 1);
for (i=0 ; i <= cnt_limit ;i++)
{
        FrameBuffer[i] = *(MessagePointerTbl[MessageIndex] + i);
}
```

Both sets of code accomplish the same end result (copying data from the LIN message buffer into the temporary transmission buffer), but there is one important difference: In the first code sample, the expression ((`MessageCountTbl[MessageIndex] & 0xF) - 1`) is evaluated each time the for () loop is executed. In the new code, the expression is evaluated one time and the loop is executed based on that value. The tradeoff is the need for one byte of stack space for the local variable `cnt_limit`. Depending upon compiler optimizations and settings, this kind of code change might even be managed automatically.

4.4.2 Configuring Messaging

The only significant detail about configuring the messaging for this design is to ensure that all transmit buffers are set up to always transmit. This is accomplished by setting the upper nibble of each entry in `MessageCountTbl[]` for a transmit buffer to 0.

4.5 Performance Issues

In this reference design, one timer module must be shared between two key time-critical functions, the LIN bus communications and the LED PWM backlighting. Performance issues can arise based on the efficiency of the application and driver code, the CPU bus speed, and the speed of the LIN bus communications. Some aspects of code efficiency have been addressed in previous sections, particularly dealing with the execution speed of the LIN driver ISR.

4.5.1 LIN Driver ISR vs. TIMCH0 Flag Polling

Because the PWM function is managed by the application software rather than by interrupts, it is necessary to ensure that the PWM is updated often enough to maintain the PWM waveform. The worst case for this is when the PWM is at the highest or lowest duty cycle settings which are not 100% or 0%. Because this is a 4-bit PWM, when `PWMValue` is 0x1 or 0xE, the PWM must be updated within about 500 µs ($1/16^{th}$ of 8 ms). Usually, this is no problem, except when LIN message traffic causes many timer interrupts to occur in this same time interval as well. Figure 4-5 shows what can happen when the PWM isn't updated properly.

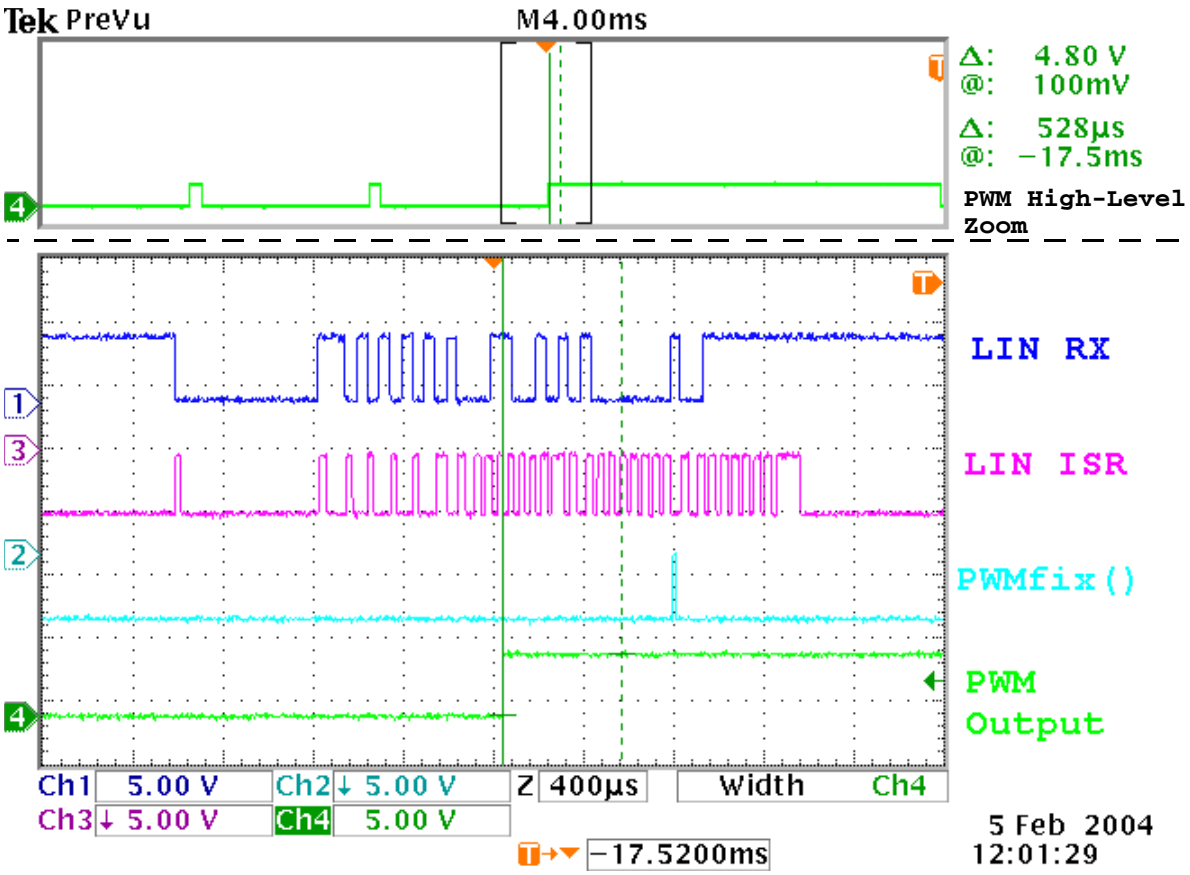


Figure 4-5. Missed PWM Update

The cursors in Figure 4-5 show how wide the PWM pulse should be (notice the two normal pulses preceding it in the high-level zoom window at the top). The ISRs for LIN communications are shown on channel 3 and the PWM update routine is shown on channel 2. In this case, the timer ISRs for the LIN communications use some of the CPU cycles that were required in the 528-µs window to allow the PWM update to occur. The result is that by the time the `PWMfix()` routine executes (channel 2), the PWM should have been low and the timer counter should have passed the target value. The PWM output stays high until the counter synchronizes. The result is an unacceptable bright flash on the LED backlighting (about 16 ms). Similarly, for the case where the duty cycle is 93.75% (`PWMValue` = 0xE), the LED turns off for the same duration.

Several issues can cause this error condition, including inefficient timer ISR code or insufficient polling of the PWM timer channel flag to

determine whether a PWM update is needed. For these reasons, the LIN timer ISR performance improvements were made and very frequent polling of TCH0F within the main switch polling loop is performed, especially at any point where a significant delay might occur within the switch polling loop. Faster LIN bus communications make the problem worse by grouping the LIN-related ISRs closer together as the edges of each bit come closer together.

Proper execution of the code can be seen in Figure 4-6. This figure shows persistence from several oscilloscope sweeps, showing several PWM cycles.

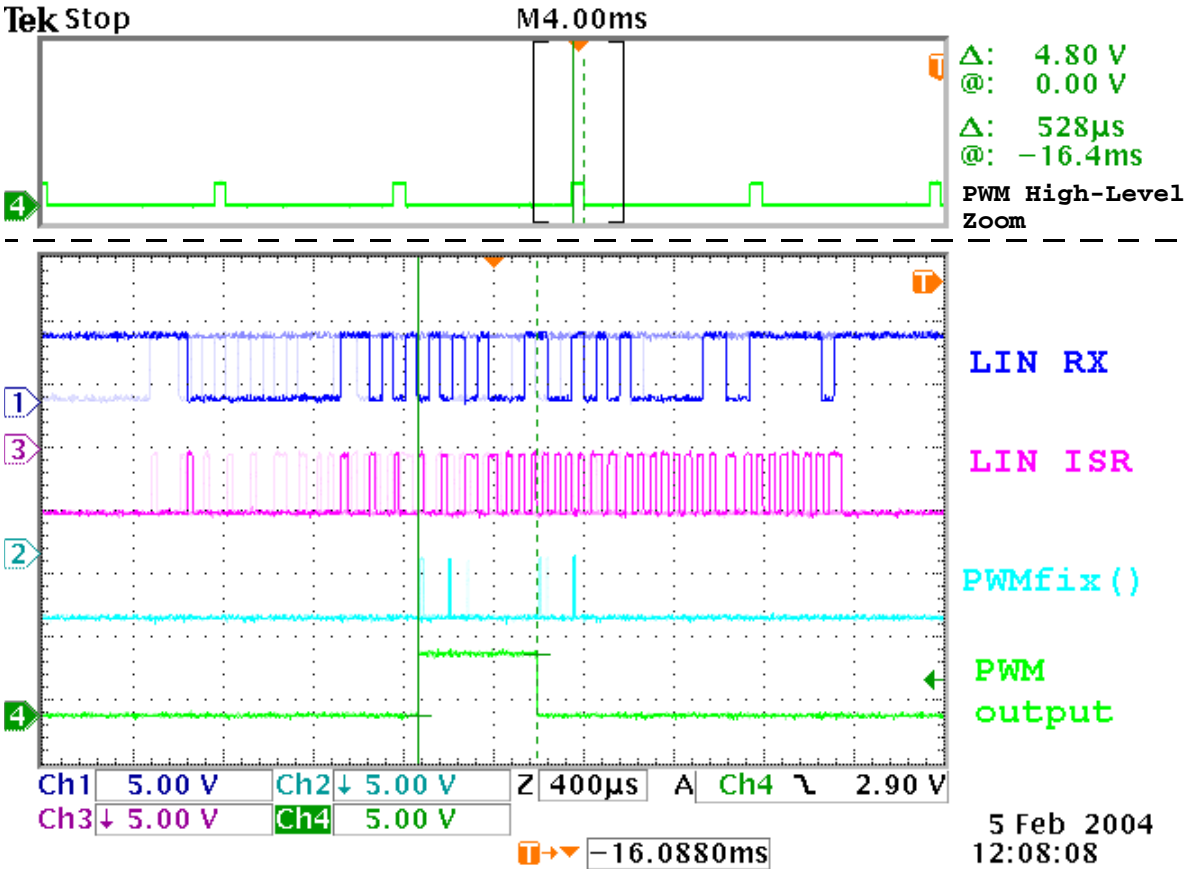


Figure 4-6. Proper Code Execution of LIN and PWM Updates

Because the LIN traffic is asynchronous to the application code, several “ghosts” of previous oscilloscope sweeps appear on both the LIN RX pin (channel 1) and the LIN ISR indicator (channel 3). Several instances of

the `PWMfix()` routine calls are also shown on channel 2. Notice that the timing of execution of this routine still varies dramatically, based on how many interrupts are being serviced for the LIN communications when the PWM needs updating. In this case, however, the code has been streamlined enough to ensure that the PWM will always get serviced in time. Other improvements can be made and are listed in Chapter 6. Design Enhancements and Upgrades.

The indicator flags shown in Figure 4-5 and Figure 4-6 were generated on pins PTB4 and PTB5 as output signals, which idle at V_{DD} and pulse low while the corresponding routine is executing. The signals were then inverted in the oscilloscope to make the resulting waveforms appear to idle low and pulse high. The reason for using the signals this way is to minimize the amount of current sourced by the MCU when the pins toggle because there is already a pullup resistor tied to the pins. These signals have been left in the code and can be activated and de-activated by enabling or disabling the `debug_flags` define statement in `main.h`.

CAUTION: *Do not press the SET or RES buttons while these flags are active. Doing so will short the output pin directly to ground with no current limiting resistor. This could cause damage to the MCU.*

Chapter 5. LIN Master Module Emulation Using the LIN Spector

5.1 Overview

The LIN Spector tool from Volcano Automotive Group is a LIN tool that can monitor and emulate LIN bus traffic. With the base version of the LIN Spector tool, an LDF file can be read in and the master node emulated to provide message headers and bus level debugging and data monitoring.

The LIN Spector tool is also capable of more advanced emulation and behavioral modeling, with the proper software add-on. For example, another slave node could be created to monitor the headlight switch and dimmer assembly, and a message could be added to the LDF file for the master to request this unit's switch data. The LIN Spector tool can then emulate the master node and this new slave node. Additionally, behavioral models of the two emulated nodes could then be programmed into a "LIN emulation control" (LEC) file. This is essentially a behavioral script for the LIN Spector tool.

A simple example is included with the download software of this reference design called `LED_Sweep.LEC`. This basic script simply varies the value of the interior brightness level (`BRIGHT_LVL`) from 0x0 through 0xF and back to 0x0 constantly. Each time the `INT_LIGHT_CMD` message is sent, the next value is transmitted. The script does not control the `LIGHT_ON` signal, so if this is 0x0, the backlighting won't turn on at all. The result of running the script is that the backlighting pulses on and off. This isn't practical in a vehicle, but does show some capabilities of using these kinds of tools to model node behavior.

The LEC file can be found in Appendix B. LED_Sweep.LEC. For details about this advanced emulation software addition, contact Volcano Automotive Group, (http://www.volcanoautomotive.com/)

Chapter 6. Design Enhancements and Upgrades

6.1 Overview

As mentioned in other sections, there are opportunities for enhancing, upgrading, and customizing this reference design. Only a brief description is offered here, as a point of departure; the designer is responsible for implementation specifics.

6.2 Software Performance Improvements

A few improvements can be made to the existing software to improve execution performance and increase the design margins of operational stability, allowing more features to be added to this code without introducing performance issues.

First, the LIN driver code can be further optimized to reduce its CPU requirements. This could be done by optimizing the existing drivers, writing new drivers, or using the MC68HC908QY4 LIN drivers outlined in AN2503/D: *Slave LIN Driver for the MC68HC908QT/QY Family*. The drivers in that application note require fewer interrupts, on average, for received messages, because they require only interrupts on each edge on the incoming data stream. One drawback of these drivers is that the LIN bus speed must be predetermined and preprogrammed rather than using the autobauding technique found in the drivers from AN2599/D.

Another simple change that does not require changing the software, when using the default reference design drivers, is to simply run the LIN bus at a slower rate. Running the bus at 9,615 bps or 10,417 bps (standard slower LIN speeds for European and US manufacturers) increases the time between LIN interrupts, allowing more application code to execute between ISRs. This increases design margin for timing.

A significant software change which can be made is to modify the `PWMfix()` routine so that if the PWM update does occur too late, the routine recognizes this condition and immediately transitions the PWM pin to the correct state. The result is that some very minor jitter might occur occasionally on one PWM cycle, but it will be approximately 1 ms rather than 16 ms based on the current design. This small amount of jitter should not be visually detectable.

6.3 Software Functionality Upgrades

A software upgrade opportunity is to complete the support for the 0x3C system sleep message and other diagnostics support. The sleep message is likely to be used by all systems and is clearly defined, so it makes sense to place support of this message directly into the driver ISR. Other diagnostics messages will vary between manufacturers, so it makes more sense to simply define these in `LINmsg.c`.

6.4 Hardware Improvements

The current hardware design could also be improved, primarily dealing with the debugging connector. One possibility is to simply remove the connector altogether, as it likely won't be used in field applications. If it is desired to keep the connector, it could use a few improvements.

Currently, if the connector is plugged in backwards, the pin normally connected to PTA2 (IRQ) would be connected to V_{DD}, which could damage the MCU, SBC, or both. To place the part in monitor mode, high voltage (V_{tst}) is applied to this pin. This is anywhere from V_{DD} +2.5 V to 9.1 V and can cause damage to components on the board. Changing this connector pinout so that PTA2 pin would line up with the no connect pin (currently pin 3) or keying the connector so it can't be plugged into backwards would be a very good improvement.

Another issue with the debug connector is that the SBC must be powered to allow a programmer to adequately drive PTA0 and PTA1 to get the MCU into monitor mode. But when the SBC has V_{bat} applied, it drives 5 V out onto V_{DD} and the programmer cannot power-on reset the

device. For debugging/development work, a cut-trace jumper on the V_{DD} connection between the MCU and the SBC would allow the developer to disconnect this during programming. This would not be a recommended practice for production units (which would be pre-programmed or ROM units anyway).

6.5 Hardware Functional Upgrades

Many opportunities exist to upgrade the hardware design to introduce new features or increase performance. Adding this keypad to a LIN bus connection provides the freedom for the designer to implement almost any feature he or she can devise. Without significantly changing the basic design, a few simple changes can be made to improve this design.

The first modification would be to change the base MCU to an MC68HC908QL4 device with the slave LIN interface controller (SLIC) module. This module is a dedicated LIN peripheral which dramatically decreases LIN driver code, reduces CPU overhead, and frees up the timer module to be used for the application. This application could easily fit into less than 2K bytes of code space on that device, allowing a smaller memory sized MC68HC908QL2 derivative to be used. The timer could then be used to drive the PWM channel or even two separate PWM channels for lighting buttons to different brightness levels. See to AN2633/D: *LIN Drivers for SLIC Module on the MC68HC908QL4* for more information and example software for that MCU (in Chapter 7. References and Acknowledgements)

Another possible change would be to use one of several multi-chip device products which contain both the MCU and the SBC. These devices combine the MCU and SBC into a single package so fewer components must be placed on the board. This also allows for smaller boards to be designed. This could also allow a larger MCU, such as the MC68HC908EY8 or MC68HC908EY16 to be designed into a smaller package, if support is needed for extra functions such as steering wheel heating and cooling or individually controlling the backlighting for each button.

Finally, a simple addition to the existing hardware design could provide for a simpler level of adaptive backlighting. Three distinct backlighting zones (for cruise control, HVAC, and radio) could easily be created with the addition of one or two discrete transistors and one additional wire to CONN1. This would also require a minor redesign of the satellite board to add one additional wire to separate its backlighting into two zones. Both zones of the satellite board backlighting could be rerouted to be driven off HS2 rather than HS1. There is no problem with driving the PWM, as both HS1 and HS2 can be PWM controlled by the same signal on the PWMin pin of the SBC. Finally, the two zones driven from HS2 could be independently gated by the discrete transistors, which are controlled via the HS3 output of the SBC and PTA2 of the MCU.

Chapter 7. References and Acknowledgements

7.1 References

Freescale Semiconductor's LIN Site: http://freescale.com/LIN

MC68HC908QY4 Product Summary Page

MC33689 Product Summary Page

AN2503/D: *Slave LIN Driver for the MC68HC908QT/QY Family*

AN2599/D: *Generic LIN Driver for MC68HC908QY4*

AN2600/D: *A Simple Keypad Using LIN with the MC68HC908QT/QY MCU*

AN2623/D: *LIN Temperature Sensor Using the MC68HC908QY/QY MCU*

AN2633/D: *LIN Drivers for SLIC Module on the MC68HC908QL4*

LIN Specification, Versions 1.3 and 2.0 from www.lin-subbus.org

VCT website: http://www.volcanoautomotive.com

7.2 Acknowledgement

Special thanks to Davor Bogavac of Freescale Semiconductor for his invaluable assistance in creating this reference design. This reference design would not have been possible without his original designs of both the hardware and the MC68HC908QY4 LIN software driver code, as well as practical advice based upon his customer-focused expertise.

Appendix A.
LIN_QY_Backlight_keypad_messaging_strategy_1_0.ldf

```
/*******************************************************************************
                                Copyright (c) Freescale Semiconductor, Inc. 2001

File Name:                LIN_QY_Backlight_keypad_messaging_strategy_0_1.ldf

Engineer:                 Matt Ruff

Location:                 OHT

Date Created:             15 December 2003

Current Revision:         0.1 -  6 Feb 2002

Notes:                    LIN QY Backlit Keypad Reference Design - LIN Description File

********************************************************************************
Freescale reserves the right to make changes without further notice to any
product herein to improve reliability, function or design. Freescale does not
assume any liability arising out of the application or use of any product,
circuit, or software described herein; neither does it convey any license
under its patent rights nor the rights of others. Freescale products are not
designed, intended, or authorized for use as components in systems  intended for
surgical implant into the body, or other applications intended to support life,
or for any other application in which the failure of the  Freescale product
could create a situation where personal injury or death may occur. Should
Buyer purchase or use Freescale products for any such unintended or
unauthorized application, Buyer shall idemnify and hold Freescale and its
officers, employees, subsidiaries, affiliates, and distributors harmless
against all claims costs, damages, and expenses, and reasonable attorney fees
arising out of, directly or indirectly, any claim of personal injury or death
associated with such unintended or unauthorized use, even if such claim alleges
that Freescale was negligent regarding the design or manufacture of the part.
Freescale and the Freescale logo* are registered trademarks of Freescale, Inc.
********************************************************************************/

LIN_description_file;
LIN_protocol_version = 1.2;
LIN_language_version = 1.2;
//LIN_speed = 4.800 kbps;
//LIN_speed = 9.615 kbps;
//LIN_speed = 10.419 kbps;
//LIN_speed = 16.525 kbps;
LIN_speed = 19.230 kbps;

//--------------------------------
Nodes {
```

```
    /*      Name        TimeBase    Jitter    */
    /*      ----        --------    ------    */
        Master: MSTR_CNTL,  1 ms,       0 ms;
        Slaves: KEYPAD;
        }
//---------------------------------
Diagnostic_addresses {//new (1.2)
        //Name: addr;
        }
/*_____________________________________________________________________Signal Definitions___*/

Signals {
    /* Name            Size   Init Sender Receiver(s)                        */
    /* ----            ----   ---- ------ -----------                        */

        /* ------------- LIN_Stat signals ----------------*/
    APP_STATE_CHG       5,      0,  KEYPAD, MSTR_CNTL;
    ERR_FIELD           3,      0,  KEYPAD, MSTR_CNTL;

        /* ------------- KEY_STATUS Cruise Control signals ----------------*/
    CC_ON               1,      0,  KEYPAD, MSTR_CNTL;
    CC_OFF              1,      0,  KEYPAD, MSTR_CNTL;
    CC_RES              1,      0,  KEYPAD, MSTR_CNTL;
    CC_SET              1,      0,  KEYPAD, MSTR_CNTL;
    CC_CST              1,      0,  KEYPAD, MSTR_CNTL;
/*      <RESERVED>              1,              0,KEYPAD, MSTR_CNTL; */
/*      <RESERVED>              1,              0,KEYPAD, MSTR_CNTL; */
    KEY_STATE_CHG       1,      0,  KEYPAD, MSTR_CNTL;

        /* ------------- KEY_STATUS Radio & HVAC signals ----------------*/
    RADIO_VOL_UP        1,      0,  KEYPAD, MSTR_CNTL;
    RADIO_VOL_DOWN      1,      0,  KEYPAD, MSTR_CNTL;
    RADIO_SOURCE        1,      0,  KEYPAD, MSTR_CNTL;
    RADIO_MEM           1,      0,  KEYPAD, MSTR_CNTL;
    HVAC_FAN_UP         1,      0,  KEYPAD, MSTR_CNTL;
    HVAC_FAN_DOWN       1,      0,  KEYPAD, MSTR_CNTL;
    HVAC_TEMP_UP        1,      0,  KEYPAD, MSTR_CNTL;
    HVAC_TEMP_DOWN      1,      0,  KEYPAD, MSTR_CNTL;

        /* ------------- INT_LIGHT_CMD signals ----------------*/
    BRIGHT_LVL          4,      0,  MSTR_CNTL, KEYPAD;
    LIGHT_ON            1,      0,  MSTR_CNTL, KEYPAD;
        }
/*_____________________________________________________________________Signal Definitions___*/

Frames {
    /* FrameName    ID      Sender   Size                                      */
    /* ---------    --      ------   ----                                     */
        LIN_Stat:        0x0A,   KEYPAD,     1 {
        /* Signal     Offset                                                  */
        /* ------     ------                                                  */
        APP_STATE_CHG,      0;
        ERR_FIELD,  5;
                }
```

```
        KEY_STATUS:       0x0B,   KEYPAD,     2 {
        /* Signal     Offset                                                */
        /* ------     ------                                                */
        CC_ON,          0;
        CC_OFF,         1;
        CC_RES,         2;
        CC_SET,         3;
        CC_CST,         4;
        /*<reserved>    5; */
        /*<reserved>    6; */
        KEY_STATE_CHG,  7;

        RADIO_VOL_UP,   8;
        RADIO_VOL_DOWN, 9;
        RADIO_SOURCE,   10;
        RADIO_MEM,      11;
        HVAC_FAN_UP,    12;
        HVAC_FAN_DOWN,  13;
        HVAC_TEMP_UP,   14;
        HVAC_TEMP_DOWN, 15;
                }

        INT_LIGHT_CMD:    0x0C,   MSTR_CNTL,  1 {
        /* Signal     Offset                                                */
        /* ------     ------                                                */
        BRIGHT_LVL,     0;
        LIGHT_ON,       7;
                }
        }
/*_____________________________________________________________________Signal Definitions___*/

Event_triggered_frames{//new (1.2)
        //EventFrame: ID, name[, name];
        }
/*_____________________________________________________________________Signal Definitions___*/

Diagnostic_frames{//new (1.2)
        MasterReq: 60{    //pub: Master
                MasterReqB0, 0;//command:0=sleep
                MasterReqB1, 8;
                MasterReqB2, 16;
                MasterReqB3, 24;
                MasterReqB4, 32;
                MasterReqB5, 40;
                MasterReqB6, 48;
                MasterReqB7, 56;
                }
        SlaveResp: 61{    //pub: any slave
                SlaveRespB0, 0;
                SlaveRespB1, 8;
                SlaveRespB2, 16;
                SlaveRespB3, 24;
                SlaveRespB4, 32;
                SlaveRespB5, 40;
                SlaveRespB6, 48;
```

```
                SlaveRespB7, 56;
                }
        }
/*_____________________________________________________________________Signal Definitions___*/

Signal_groups {
        //GroupName: Size {
        //      SignalName, offset;
        //      }
        }
/*_____________________________________________________________________Signal Definitions___*/

Schedule_tables {

        FAST_POLL {
        LIN_Stat    delay   20.00   ms;     // 10ms for 9600 bps
            }
        CHECK_SWITCHES {
        KEY_STATUS  delay   21.00   ms;     // 11ms for 9600 bps
            }
        LIGHTS_ONLY {
        INT_LIGHT_CMD  delay   15.00   ms;     // 11ms for 9600 bps
            }
        CHECK_SWITCHES_AND_LIGHT {
        KEY_STATUS  delay   15.00   ms;     // 11ms for 9600 bps
        INT_LIGHT_CMD  delay   15.00   ms;     // 11ms for 9600 bps
            }

        }
/*_____________________________________________________________________Signal Definitions___*/

Signal_encoding_types {
        //Name {
        //      logical_value, signal value, "textinfo"
        //      physical_value, min value, max value, offset, scale, "textinfo"
        //      bcd_value ;
        //      ASCII_value ;
        //      }

        APP_STATE_CHG_field {
                logical_value, 0, "No Error";
                logical_value, 1, "Key was pressed";
                logical_value, 2, "RESERVED";
                logical_value, 3, "RESERVED";
        }

        LIN_Error_Field {
                logical_value, 0, "No Error";
                logical_value, 1, "Reset";
                logical_value, 2, "RESERVED";
                logical_value, 3, "RESERVED";

                logical_value, 4, "Bit-Error";
                logical_value, 5, "Checksum-Err";
                logical_value, 6, "Byte Framing Error";
```

```
                logical_value, 7, "ID-Parity-Error";
        }

    Switch_Pos    {
        logical_value, 0, "OPEN";
        logical_value, 1, "PRESSED";
        }
        Switch_Stat_Hist {
                logical_value, 0, "No change";
                logical_value, 1, "Key was pressed";
        }

        Light_Level {
                physical_value, 0, 16, 6.65, 0, " %";
        }

    Light_Status    {
        logical_value, 0, "OFF";
        logical_value, 1, "ON";
        }

        }
/*_____________________________________________________________________Signal Definitions___*/

Signal_representation {
        //EncName: SignalName [, SignalName];

    APP_STATE_CHG_field:    APP_STATE_CHG;
        LIN_Error_Field:        ERR_FIELD;

    Switch_Pos: CC_ON, CC_OFF, CC_RES, CC_SET, CC_CST,
                RADIO_VOL_UP, RADIO_VOL_DOWN, RADIO_SOURCE, RADIO_MEM,
                HVAC_FAN_UP, HVAC_FAN_DOWN, HVAC_TEMP_UP, HVAC_TEMP_DOWN;

        Switch_Stat_Hist: KEY_STATE_CHG;

        Light_Level:    BRIGHT_LVL;
        Light_Status:   LIGHT_ON;
        }
/*_____________________________________________________________________Signal Definitions___*/
```

Appendix B. LED_Sweep.LEC

```
// This is a LIN emulator control file
// Created by LINspector on 1-20-2004

LIN_emulation_control_file;
LIN_protocol_version = 1.2;
LIN_language_version = 1.2;
LIN_description_file = \\mecd-ra5782-l1\d$\Data\MUX_Information\LIN\Reference Design
Activities\QY LIN Slave - Davor Bogavac\Messaging
Strategy\LIN_QY_Backlight_keypad_messaging_strategy_0_1.ldf ;

Emulated_nodes
        {
        MSTR_CNTL;
        }

Control_program
{
// Defines:
// #define

// Variables:
// uint16
uint16 count;
uint16 i;

// Initializations:

// Main cycle:
while(1)
        {
        for(count= 0; count<16; count++)
                {
                BRIGHT_LVL = count;
                while(!test_flag(INT_LIGHT_CMD.RECEIVED))
                          {
                    }//On INT_LIGHT_CMD.RECEIVED

                clear_flag(INT_LIGHT_CMD.RECEIVED);
                // Wait for complete, then roll to next cnt
                //              for(i=0;i<35;i++){;;}
                }

        for(count= 14; count>0; count--)
                {
        BRIGHT_LVL = count;
                while(!test_flag(INT_LIGHT_CMD.RECEIVED))
```

```
                        {
                    }//On INT_LIGHT_CMD.RECEIVED

                clear_flag(INT_LIGHT_CMD.RECEIVED);
                // Wait for complete, then roll to next cnt
                //              for(i=0;i<35;i++){;;}
                }
        }       // while(1)

}       // Control program
```

Appendix C. Schematic and BOM

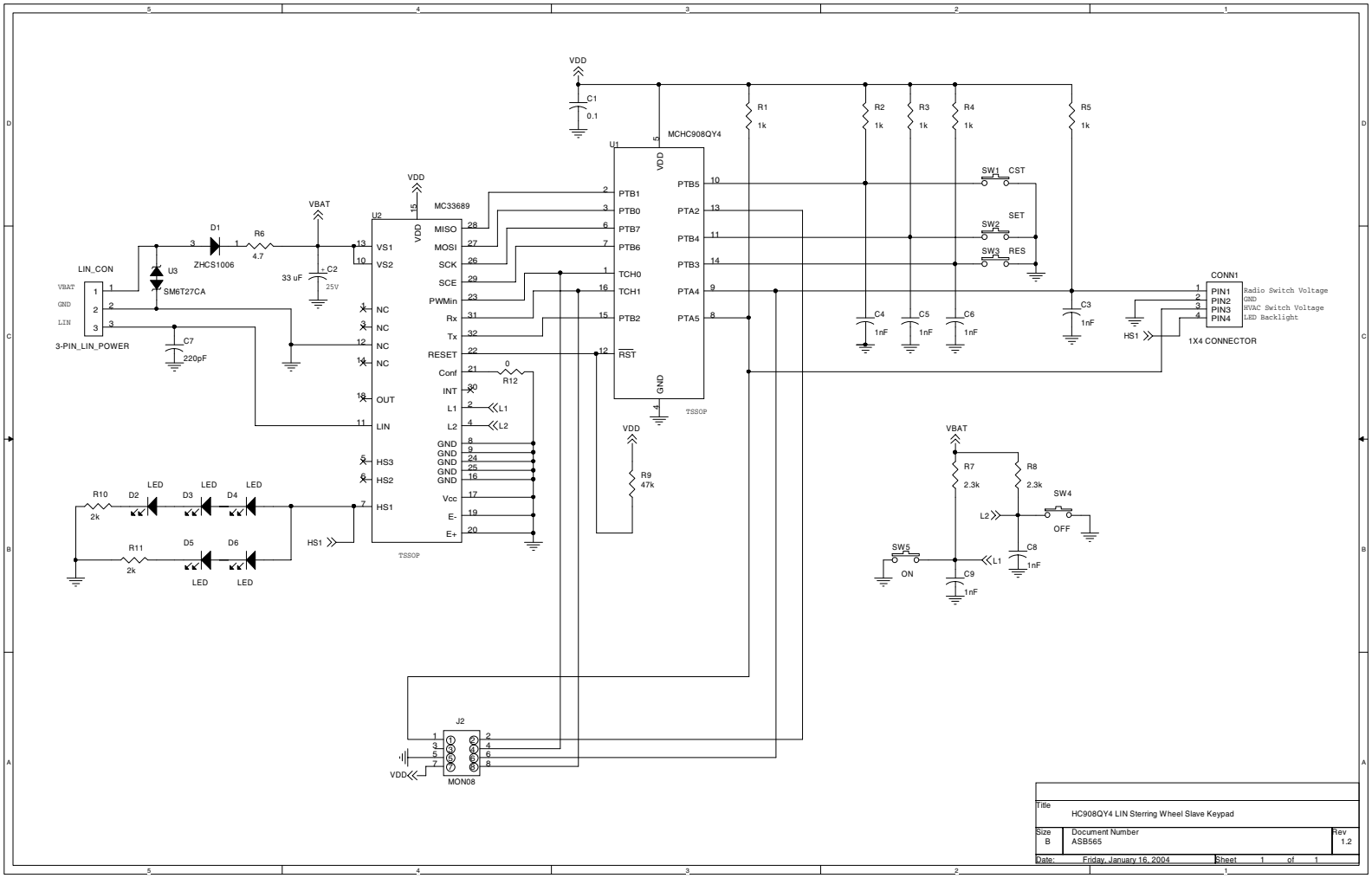
Bill of Materials (BOM)

Designators	Qty	Description	Vendor	Part Number
C1	1	0.1 uF Capicator (0603)	Digi-Key	PCC2277CT-ND
C2	1	33 uF Capicator 25V (7343)	Digi-Key	P11298CT-ND
C3,C4,C5,C6,C8,C9	6	1 nF Capicator (0603)	Digi-Key	PCC2151CT-ND
C7	1	220 pF Capicator (0603)	Digi-Key	PCC221ACVCT-ND
D1	1	MIF60 Diode (SOT23)	Digi-Key	ZHCS1006CT-ND
D2,D3,D4,D5,D6	5	Red LED	Digi-Key	67-1371-1-ND
CONN1	1	1X4 .1oc Header See Note 1 and Note 2	Digi-Key	S2011-36-ND
J2	1	Mon08 Connector 2x4 .1oc Header	Digi-Key	WM18203-ND
LIN_CON	1	1X3 .1oc Header See Note 1 and Note 2	Digi-Key	See CONN1
R1,R2,R3,R4,R5	5	1k Ohm 5% Resistor (0603)	Digi-Key	311-1.0KGCT-ND
R6	1	4.7 Ohm Resistor (0603)	Digi-Key	311-4.7GCT-ND
R7,R8	2	2.32K Ohm Resistor (0603)	Digi-Key	P2.32KHCT-ND
R9	1	47K Ohm 5%Resistor (0603)	Digi-Key	311-47KGCT-ND
R10,R11	2	2K ohm 1% Resistor (0805)	Digi-Key	P2.00KCCT-ND
R12	1	0.0 Ohm 5% Resistor	Digi-Key	311-0.0GCT-ND
U1	1	HC908QY4 - Nitron	Digi-Key	MC68HLC908QY4MDT
U2	1	MC33689 - LIN System Basis Chip	Digi-Key	PC33689DW
U3	1	SM6T27CA (DO214)	Mouser Electronics	511-SM6T24CA
ASB525 LIN Keyboard bare PC board	1	Bare PC board	DS Electronics	ASB525
Note 1: Shipped in strips of 36x2. Cut to length. Note 2: Constructed from a 2x4 connector with one pin removed				

WIRE HARNESS COMPONENTS				
RIGHT-Harness-Conn	1	C-Grid SL .100 Pocket Header Mating Connector Housing "G" Version, 1x4	Digi-Key	WM2902-ND
	4	Crimp Terminals -	Digi-Key	
LEFT-Harness-Conn	1	C-Grid Crimp Connector Housing Mating Connector Housing "A" Version, 2x4	Digi-Key	WM2521-ND
	7	Crimp Terminals -	Digi-Key	
CLOCKSPRING-Harness-Conn	1	C-Grid SL .100 Pocket Header Mating Connector Housing "G" Version, 1x3	Digi-Key	WM2901-ND
	3	Crimp Terminals -	Digi-Key	WM2512-ND
CLOCKSPRING-Conn	1	C-Grid SL .100 Pocket Header Straight Header	Digi-Key	WM4801-ND
ALTERNATE LIN MASTER Connector (Motorola LIN Tools Connector)	1	CONN RECEPT 4POS VERT DUAL	Digi-Key	WM3701-ND
	1	CONN TERM FEMALE 18-24AWG TIN	Digi-Key	WM2501-ND
Debugging Harness	1	Crimp Housing	Digi-Key	WM18032-ND
	8	Crimp Terminal	Digi-Key	WM18056-ND

2003 Ford Expedition Keypad (Eddie Bauer Edition)	1	NL SW AS	Ford Motor Company	1L2Z 9C888 BA

Schematic


VDD
C1
0.1
U1
MCHC908QY4
R1
1k
R2
1k
R3
1k
R4
1k
R5
1k
SW1 CST
SW2 SET
SW3 RES
C4
1nF
C5
1nF
C6
1nF
C3
1nF
CONN1
PIN1 Radio Switch Voltage
PIN2 GND
PIN3 HVAC Switch Voltage
PIN4 LED Backlight
1X4 CONNECTOR
HS1
VBAT
D1
ZHCS1006
R6
4.7
C2
33 uF
25V
U3
SM6T27CA
LIN_CON
3-PIN_LIN_POWER
C7
220pF
U2
MC33689
VS1
VS2
NC
OUT
LIN
HS3
HS2
HS1
MISO
MOSI
SCK
SCE
PWMin
Rx
Tx
RESET
Conf
INT
L1
L2
GND
Vcc
E-
E+
TSSOP
R12
0
R9
47k
PTB1
PTB0
PTB7
PTB6
TCH0
TCH1
PTB2
RST
PTB5
PTA2
PTB4
PTB3
PTA4
PTA5
R10
2k
R11
2k
D2
D3
D4
D5
D6
LED
R7
2.3k
R8
2.3k
SW4
OFF
SW5
ON
C8
1nF
C9
1nF
J2
MON08
Title
HC908QY4 LIN Sterring Wheel Slave Keypad
Size
B
Document Number
ASB565
Rev
1.2
Date:
Friday, January 16, 2004
Sheet
1
of
1

How to Reach Us:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale Semiconductor products, please visit
http://www.freescale.com

DRM058
Rev. 1, 7/2004

freescale™
semiconductor