Order this document
by AN1816/D

AN1816

USING THE HC912B32 TO IMPLEMENT THE DISTRIBUTED SYSTEMS INTERFACE (DSI) PROTOCOL

By Tracy McHenry

August 1999

Introduction

System design requirements are continually changing as systems become increasingly complex. In conventional systems where sensors and actuators are connected directly to a microcontroller (MCU), the number of pins available on the MCU limits system expansion. As a result, extra costs can be incurred if another MCU is required or if the hardware has to be redesigned to accommodate a higher pin count MCU. An alternative approach is to move to a distributed bus architecture. This option allows one master MCU to interconnect to many remote sensors and actuators.

The Distributed Systems Interface (DSI) is one such master/slave system, with the central control module being the master and the remote sensors and actuators acting as slave devices. A key feature of the DSI architecture is that the sensors and actuators can be connected on the same bus. Another benefit of the DSI is that it promotes the use of standard components and interfaces which enables maximum re-use and accelerates time to market. This is a great advantage to systems designers who are therefore able to develop flexible system solutions. It was initially developed for the automotive market although it is equally suited to other applications that require distributed sensors and/or actuators. Examples of possible DSI applications include occupant safety systems, body networks, building and industrial controls.

This application note provides an overview of the DSI and describes the hardware and software design of a demonstrator system.

Application Note

Distributed Systems Interface (DSI) Overview

The Distributed Systems Interface (DSI) was designed to interconnect multiple remote sensor and actuator devices to a central control module. It provides simultaneous support for sensors and actuators using a 2-wire bus that provides both power and communications. This results in savings in wiring costs and connector complexity. Unlike conventional systems, the size of the connector for the control module does not need to grow to accommodate every new sensor or actuator. A new sensor or actuator can be added to the bus without reconfiguring the system design. The DSI, therefore, enables the development of easily expandible systems.

The DSI communication bus is simple yet robust. Signals are superimposed onto the power line and, as communication between the master and slave nodes is bi-directional, the DSI makes efficient use of bus bandwidth. Also, to ensure message integrity, each message contains a 4-bit Cyclic Redundancy Check (CRC).

Slave nodes can be attached to the bus in daisy chain or parallel connections. Each slave device on a bus has a unique address. The daisy chain connection allows the central module to establish the node addresses at power-up. The parallel configuration can be used for devices that have pre-programmed or fixed addresses. It is possible to have a combination of the two on one bus with the maximum number of nodes on a DSI bus being 16 (1 master and 15 slaves).

Signalling

The DSI uses two mediums for signalling - voltage mode for messages from the master to the slaves, and current mode for the responses from the slaves.

Voltage Mode Encoding

The voltage mode signal bits are sent on a 2:1 ratio as shown in Figure 1 Voltage Mode Bit Encoding.

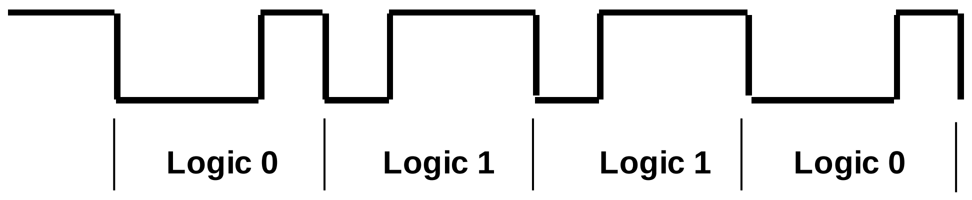

Figure 1 Voltage Mode Bit Encoding

The DSI protocol has been designed such that the first third of a voltage mode signal bit is always low and the last third is always high. The central third defines the signal value. For a logic zero the master produces a signal that is low for 2/3 of the bit time and high for the final 1/3. For a logic one the signal is low for 1/3 of the bit time and high for 2/3 of the bit time. Each slave has a built-in oscillator. This feature of the DSI message protocol has created a high immunity to temporal distortion. Consequently, each slave IC is simpler and cheaper to design while still able to capture all messages accurately. An added benefit is that it allows for a range of operating frequencies.

Bus Voltage Levels

The voltage mode signalling uses a tri-level bus as shown in Figure 2 Tri-level bus.

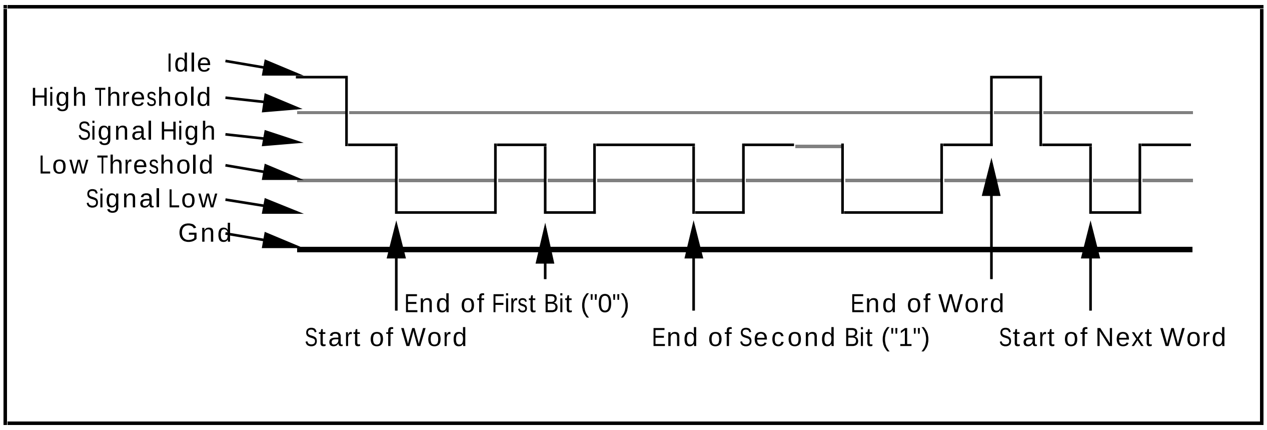

Figure 2 Tri-level bus

The DSI bus provides power to the slave nodes as well as supporting bi-directional communication between the slave nodes and the master. When the bus is at the idle voltage, which ranges from 8 to 25 volts, it supplies power to the slave nodes. A high threshold level (typically 6V) and a low threshold level (typically 3V) play a significant part in message transmission. The start of a word occurs when the bus voltage drops from the idle voltage below the high threshold level and then below the low threshold level. Data values are determined by the ratio of time spent above and below the low threshold. The voltage rising above the high threshold level signals the end of a word.

Current Mode Encoding

Slave responses to commands are returned by modulating the amount of current sunk by the device. This is measured at the end of each bit to determine a 'zero' or 'one' response. When responding with a logic one, the slave draws additional current from the source during the bit time. Conversely, no extra current is drawn when responding with a logic zero. Figure 3 Current Waveform shows a representation of the current waveform referenced to a voltage waveform.

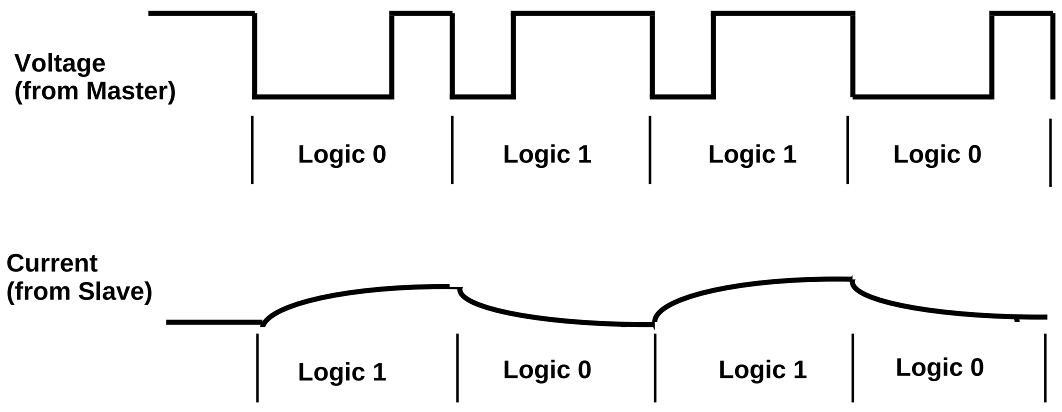

Figure 3 Current Waveform

Message Format

DSI messages are composed of individual words separated by a minimum frame delay. Transfers are full duplex, that is, command messages from the master occur at the same time as responses from the slaves. This is an important feature of the DSI as it doubles the effective signal bandwidth. Slave responses to commands occur during the next command message. This allows slaves time to decode the command, act upon it and then respond to the master. The minimum frame delay is present to allow recharge of energy storage devices in the slaves. This is necessary because the slave receives its power from the signal line.

Message Encoding

Message encoding depends on the direction of the transfer. Command and control messages are sent from the master to the slave. Response messages are sent from the slaves to the master. In both cases there are long word and short word messages. A long word consists of 16-bits of information followed by a 4-bit cyclic redundancy check (CRC). A short word is composed of 8-bits of information followed by the 4-bit CRC.

Command and Control Messages

The long word command and control message encoding is shown in Figure 4 Long Word Command and Control Message.

DATA								ADDRESS				COMMAND				CRC			
D7	D6	D5	D4	D3	D2	D1	D0	A3	A2	A1	A0	C3	C2	C1	C0	X3	X2	X1	X0

Figure 4 Long Word Command and Control Message

The message consists of 8 bits of data, the encoded 4-bit address of the intended slave device, a 4-bit encoded command, and the calculated 4-bit CRC.

The short word command and control message encoding is shown in Figure 5 Short Word Command and Control Message.

ADDRESS				COMMAND				CRC			
A3	A2	A1	A0	C3	C2	C1	C0	X3	X2	X1	X0

Figure 5 Short Word Command and Control Message

The message consists of the encoded 4-bit address of the intended slave device, a 4-bit encoded command, and the calculated 4-bit CRC.

Response Messages

A long word response message is sent from the slave to the master in response to a long word command and control message to the slave's address. The response is transmitted during the next command and control message. The long word response message encoding is shown in Figure 6 Long Word Response Message.

DATA BYTE 1								DATA BYTE 2								CRC			
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	X3	X2	X1	X0

Figure 6 Long Word Response Message

The message consists of two 8-bit data bytes and the calculated 4-bit CRC

A short word response message is sent from the slave to the master in response to a short word command and control message to the slave's address. The response is transmitted during the next command and control message. The short word response message encoding is shown in Figure 7 Short Word Response Message.

DATA								CRC			
D7	D6	D5	D4	D3	D2	D1	D0	X3	X2	X1	X0

Figure 7 Short Word Response Message

The message consists of one 8-bit data byte and the calculated 4-bit CRC.

Long words are always sent in response to long word commands and short words are always sent in response to short word commands. When the word format changes between successive commands, the first response sent during the new format will be invalid since it will not have the proper number of bits.

Error Checking

The master and slaves calculate a CRC on the information portion of their received messages. The message is valid only if the calculated CRC matches the CRC sent as part of the message. An error bit is set in the master when it receives an invalid message. The slaves discard and ignore all invalid received messages and in addition do not respond to them.

Slave Device Addressing

Each slave device on the bus must be given a unique 4-bit address (from 1 (0001) to 15 (1111)) and assigned to one of four groups. Address 0 (0000) is used to address all 15 slave nodes at once.

Programmable Devices

After system power up the master must set the address of all daisy chain slave devices with programmable addresses before network communications can commence. These devices have a bus switch on the power/signal line. At power up the programmable device bus switches are open and only close once an initialisation message has been received.

With the first bus switch open, the bus only goes as far as the first slave. When the master sends a slave initialisation command, the first slave device stores its address information and closes its bus switch. The second daisy chain slave is now connected to the network. When the master initialises the second slave's address, the first device responds with an initialisation response message. The response message echoes the programming information back to the master so that it knows that the address was successfully established. Each slave sends the initialisation response message only once after receiving a program address command message.

An advantage of having programmable devices present in the system is that they may be replaced and/or the system may be reconfigured by adding nodes to the bus and the system will automatically reconfigure itself at the next power up.

Pre-programmed Devices

Slaves with pre-programmed addresses do not require a bus switch. On power-up the stored pre-programmed address becomes the slave address. However, pre-programmed devices must still receive an Initialisation Command and reply with an Initialisation Response before responding to any other bus commands.

HARDWARE DESIGN

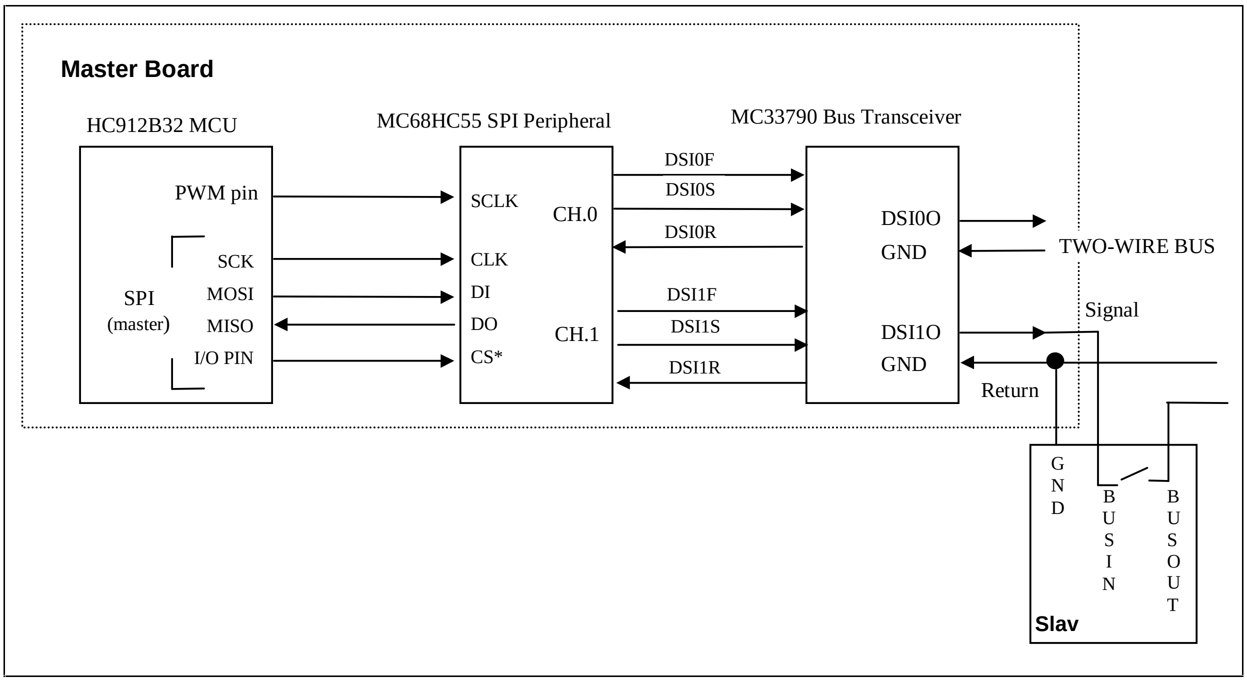

Figure 8 Top Level DSI System Connections

Several ICs are used in the development of the system :

- Master board
  – HC912B32
  – MC68HC55 SPI Peripheral
  – MC33790 Bus Transceiver
- Slave board
  – evaluation slave device, the BEM IC

Master Board

Figure 8 Top Level DSI System Connections shows how the ICs are connected together. It should be noted that the diagram shows a daisy chain connected slave but slave nodes can also be connected onto the bus in parallel. The parallel configuration is used for slave nodes that have pre-programmed addresses. The SPI of the HC912B32 is set up such that it acts as the master and all communication from the HC912B32 to the MC68HC55 is via the SPI. The MC33790, the physical layer interface to the DSI bus, sends commands to and receives responses from the slave devices.

Software required to control the system can be programmed into the HC912B32's 32k of FLASH memory. It also has 768 bytes of EEPROM and 1k of RAM. The PWM is used to provide the MC68HC55 with a system clock. It is set up such that PWM channel 0 is output on port P pin 0 (PP0) and connected to the SCLK pin of the MC68HC55 SPI Peripheral. The HC912B32's on board SPI is a key element in the DSI communication protocol and is set up as follows. The MOSI, MISO and SCLK pins on the HC912B32 are connected to the MC68HC55's DI, DO, CLK pins respectively. Port S pin 7 (PS7) is connected to the $\overline{CS}$ pin on the MC68HC55. It should then be defined in the software as a general-purpose I/O pin and set up to drive $\overline{CS}$ on the MC68HC55. $\overline{CS}$ is an active low signal that is controlled by the HC912B32. When driven low it indicates the start of message transmission from the HC912B32 to the MC68HC55 and in turn to the MC33790 and then to the slave nodes. This is the start of what is termed a SPI burst transfer (refer to the sub section: 'Initialisation of the PWM and SPI' in the Software Design section of this Application Note). The end of a SPI burst transfer is signalled by the HC912B32 pulling $\overline{CS}$ high. Commands are sent to and responses are received from the slave nodes via the SPI during a burst transfer. The data written to the SPI data register is transferred into the MC68HC55's data register, which transmits the message to the slave nodes via the MC33790 Bus Transceiver.

The MC68HC55 is the protocol controller of the system and controls all the digital functions of the DSI. It contains 2 independent DSI channels, each capable of interfacing to up to 15 slave nodes. The MC68HC55 SPI Peripheral uses 3 pins to transmit a message to the MC33790 Bus Transceiver. Pin DSIxS (signal) on the MC68HC55 transmits the data output signal to the MC33790. Data bits on this signal line are pulse length encoded voltage levels. A logic zero starts with a falling edge on DSIxS and is low for two thirds of the bit time and then high for one third of the bit time. A logic one starts with a falling edge on DSIxS and is low for one third of the bit time and then high for two thirds of the bit time. DSIxF (frame) output pin idles high and is driven low during each transfer frame. DSIxR is the data input signal from the MC33790. The MC68HC55 samples the CMOS level on this pin at the end of a bit time. This level corresponds to the current sensed by the MC33790.

The MC33790 Bus Transceiver is an analogue SmartMOS™ device, which serves as the physical interface to the two-wire DSI bus. It uses a combination of pulse length encoded voltage levels for transmit data to slave nodes and current return signals for receive data at the same time.

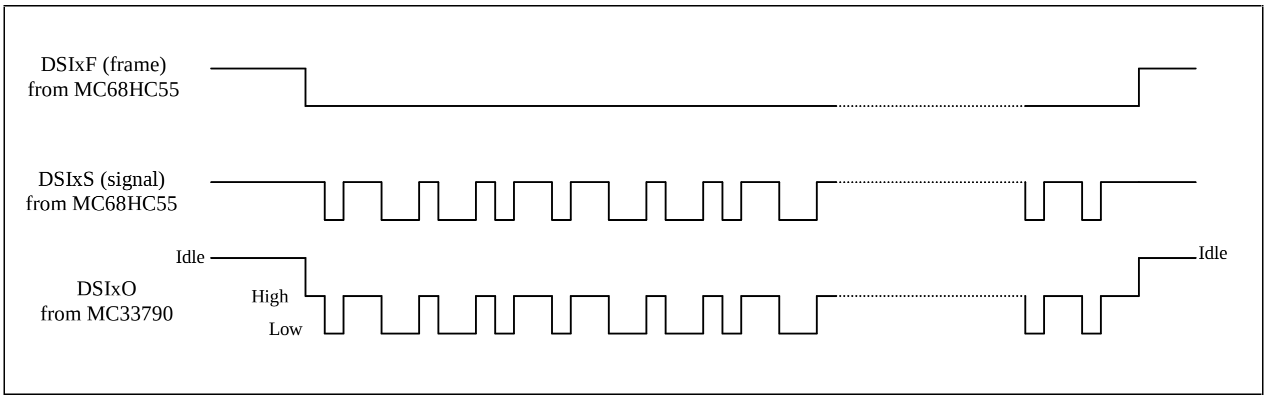

Figure 9 MC68HC55 frame and output signals with respect to MC33790 DSIx0 signal

When the MC33790 receives data for transmission to the slave nodes from the MC68HC55, its bus transmitter circuit converts the 0 to 5 volt inputs from the MC68HC55's DSIxF (frame) and DSIxS (signal) to a voltage level on the output DSIxO which connects to BUS_IN of the first slave node. The value output on DSIxO of the MC33790 depends on the values of DSIxF and DSIxS as shown in Table 1 DSIxO Truth Table. See also Figure 9 MC68HC55 frame and output signals with respect to MC33790 DSIx0 signal.

DSIxF	DSIxS	DSIxR	DSIxO
0	0	Return Data	Low (1.5V)
0	1	Return Data	High (4.5V)
1	0	0	High Impedance
1	1	0	Idle

Table 1 DSIxO Truth Table

When the MC33790 is receiving data from the slave nodes, it samples the current responses on the rising edge of the final third of bit time. The current response is then compared against a reference point, which determines whether a low or high has been returned.

Schematics and Layout Considerations - Master Board

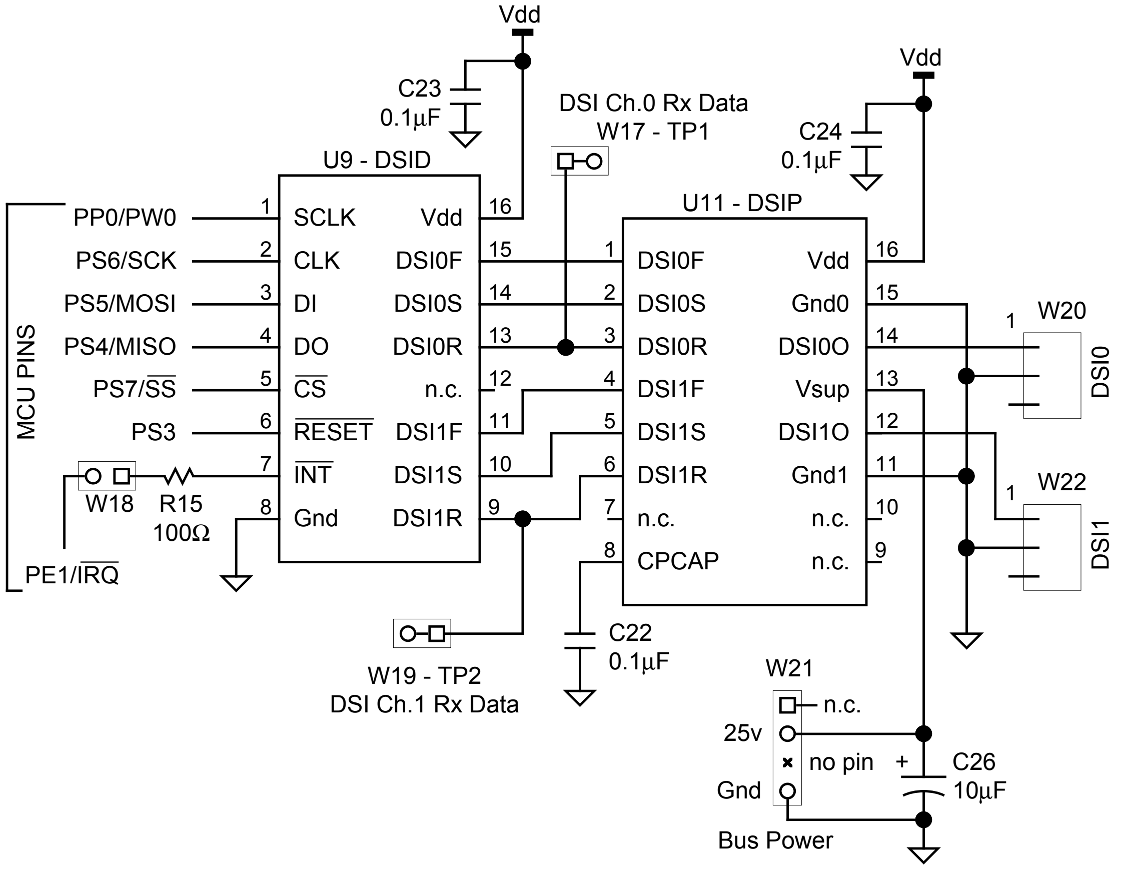

Figure 10 Master Board Schematic

Figure 10 Master Board Schematic shows the schematic for the master board. The board was based on an existing HC912B32 Evaluation Board (EVB) with the addition of the required DSI circuitry. The HC912B32 EVB circuitry allows the board to be used to evaluate prototype hardware and software. When laying out the DSI circuitry on the master board, the bus traces out of the MC33790 were made as wide as possible to deal with the presence of the higher voltage (the DSI bus maximum idle voltage is 25V). All components were positioned where clock and bus trace lengths could be kept to a minimum. 0.1µF capacitors were used to filter the bus supply voltage and to decouple the power supplies to the MC68HC55 and MC33790. These capacitors were placed as close to the ICs as possible.

Slave Board

The slave board discussed in this application note is one that was designed as part of a DSI evaluation tool kit. It contains the Bus Evaluation Module (BEM) IC, which is a slave DSI interface evaluation device. It provides bi-directional communications from the DSI bus. The slave board contains an on board potentiometer which can be used to vary the voltage input to the BEM IC. This voltage is converted by the BEM IC to a digital signal that is then transmitted over the DSI bus. Alternatively, an accelerometer or other analogue output device could be connected using the wire wrap area of the slave board and the jumper setting changed such that the output of the accelerometer is input to the BEM IC. Multiple slave nodes can be attached to one DSI bus by daisy chaining them together.

Schematics and Layout Considerations - Slave Board

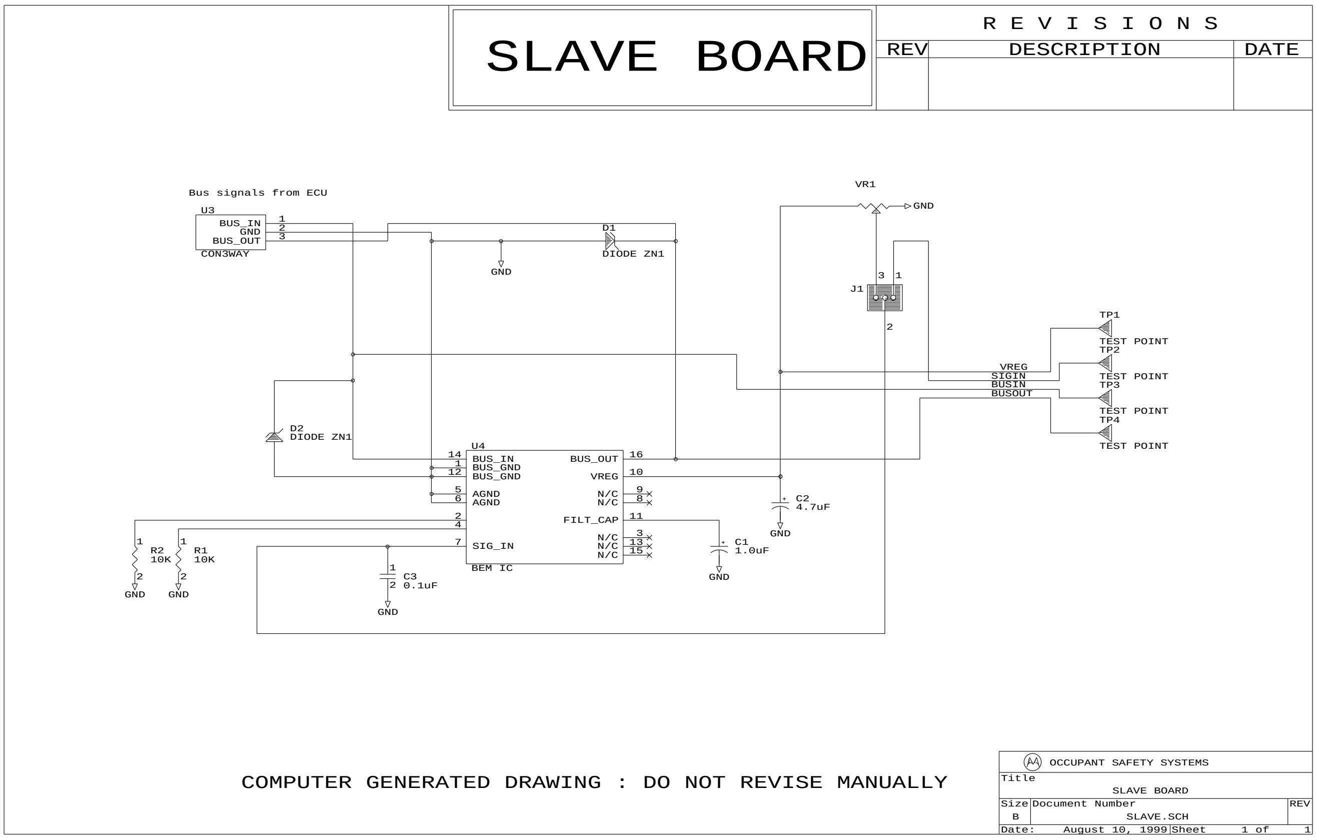

Figure 11 Slave Board Schematic

Figure 11 Slave Board Schematic shows the schematic for the slave board. A benefit of the DSI architecture is that the slave board requires only a few additional components. Zener diodes are used to protect against ESD damage to signals BusIn and BusOut. C1 (filt_cap) supplies the power to the BEM IC during signalling. When laying out the slave board, the pads for capacitor, C1 were enlarged so that they could accommodate various values of capacitor from 1µF up to 4.7µF depending on what was required of the evaluation system. For the slave node described in this Application Note a 1µF capacitor was selected as being capable of storing enough charge to power the BEM IC during signalling. All components were positioned where signal trace lengths could be kept to a minimum and signal traces were made as wide as possible. Also, analogue and digital grounds were connected together as close to the BEM IC as possible.

SOFTWARE DESIGN

The initialisation software can be divided into 3 sections - initialisation of the PWM and SPI, initialisation of the MC68HC55 SPI peripheral's registers and initialisation of the slave nodes.

Initialisation of the PWM and SPI

The PWM on the HC912B32 provides the system clock (SCLK) for the MC68HC55 SPI Peripheral. The system software must initialise the PWM so that it supplies a clock signal to the MC68HC55 SPI Peripheral with the appropriate duty cycle and period. An example of possible C source code that can be used to perform this set-up is shown in function InitPWM in Appendix 1 - Source Code. The PWM registers, in this example, are set up to generate a clock signal with a duty cycle of 50% and a period of 3.5µs (frequency of 285kHz).

The SPI is set up such that the slave select ($\overline{\text{SS}}$) pin on the HC912B32 (connected to chip select ($\overline{\text{CS}}$) on the MC68HC55 SPI Peripheral) is configured as a general purpose I/O pin allowing the software to control it. This is necessary for SPI Burst Transfers, thus enabling the HC912B32 to communicate with the slave nodes via the MC68HC55 and the MC33790. Figure 12 SPI Burst Transfer Example shows an example of an SPI burst transfer.

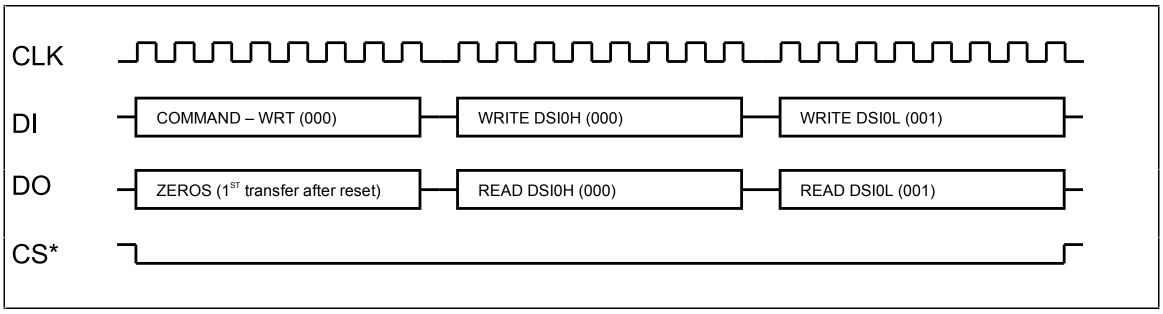

Figure 12 SPI Burst Transfer Example

When $\overline{CS}$ is driven low by the HC912B32, the MC68HC55 SPI Peripheral is enabled and the first SPI transfer after this is a command transfer. Bit7 of this command determines if it is a read (0) or a write (1) command. Bits[2:0] specify the address of one of the eight MC68HC55 registers and an internal pointer is established. Data sent back to the HC912B32 from the MC68HC55 during a command transfer is read data from the adress previously pointed to (this would be all zeros for the first transfer after reset). Any additional transfers result in a write-to or read-from successive registers in the MC68HC55. The internal register pointer is incremented at the end of the transfer and will roll over from 7 (111) to 0(000). $\overline{CS}$ remains low throughout the whole burst sequence and is driven high at the end by the HC912B32. Possible software routines to perform the SPI set-up and to control the transfer of data in SPI burst mode are shown in the source code detailed in functions InitSpi and SpiBurst in Appendix 1 - Source Code.

Initialisation of the MC68HC55 SPI Peripheral's Registers

Table 2 DSI Registers shows the MC68HC55 registers. The HC912B32 can read-from or write-to the MC68HC55's seven registers through the SPI interface. Data to be transferred to the slave nodes is written into the

DSI0H - Data Access for DSI/D Channel 0 (high byte) address - **000**

Bit 7	6	5	4	3	2	1	Bit 0	
Bit-15	Bit-14	Bit-13	Bit-12	Bit-11	Bit-10	Bit-9	Bit-8	High

DSI0L- Data Access for DSI/D Channel 0 (low byte) address - **001**

Bit 7	6	5	4	3	2	1	Bit 0	
Bit-7	Bit-6	Bit-5	Bit-4	Bit-3	Bit-2	Bit-1	Bit-0	Low

Table 2 DSI Registers

DSI1H - Data Access for DSI/D Channel 1(high byte) address - **010**

Bit 7	6	5	4	3	2	1	Bit 0	
Bit-15	Bit-14	Bit-13	Bit-12	Bit-11	Bit-10	Bit-9	Bit-8	High

DSI1L- Data Access for DSI/D Channel 1(low byte) address - **011**

Bit 7	6	5	4	3	2	1	Bit 0	
Bit-7	Bit-6	Bit-5	Bit-4	Bit-3	Bit-2	Bit-1	Bit-0	Low

DSISTAT- DSI Status Register address - **100**

Bit 7	6	5	4	3	2	1	Bit 0
ER1	TFE1	TFNF1	RFNE1	ER0	TFE0	TFNF0	RFNE0

DSI0CTRL- DSI Channel 0 Control Register address - **101**

Bit 7	6	5	4	3	2	1	Bit 0
CDIV0B	CDIV0A	DLY0B	DLY0A	RIE0	TIE0	0	MS0

DSI1CTRL- DSI Channel 1 Control Register address - **110**

Bit 7	6	5	4	3	2	1	Bit 0
CDIV1B	CDIV1A	DLY1B	DLY1A	RIE1	TIE1	0	MS1

DSIENABL address - **111**

Bit 7	6	5	4	3	2	1	Bit 0
0	0	0	0	0	0	EN1	EN0

Table 2 DSI Registers (Continued)

DSIxH:DSIxL register pair for 16-bit messages; DSIxL for 8-bit. The transfer begins once DSIxL has been written to. Similarly, responses from the slave nodes are written into these registers. The DSISTAT register provides status information and should be checked before and after transfers. It contains an error flag, ERx, which indicates if the master received an invalid CRC value. Also, transmit and receive status information, TFEx, TFNFx and RFNEx, is available through this register. The DSIxCTRL register is written to before data is sent over the DSI bus. This register is used to specify an additional divider between the SCLK input and the bit timing circuitry, CDIVx[B:A], as well as defining the interframe delay, DLYx[B:A]. It is also used to enable interrupts, RIEx and TIEx, and to select 12 -bit (8 data bits plus 4 CRC bits) or 20-bit (16 data bits plus 4 CRC bits) messages, MSx. This register is updated as soon as new data is received over the SPI interface, however, the new value does not take affect until the next DSI clock cycle after the conclusion of the SPI write to this register. Finally, the DSIENABL register is used to enable or disable each DSI channel. The function SetupDSI shown in the source code in Appendix 1 - Source Code is an example of how to set up the MC68HC55 SPI Peripheral's registers. It uses the SPI burst routine discussed in sub section: 'Initialisation of the PWM and SPI' and sets up the registers for initialisation of the programmable slave nodes.

Initialising the Slave Nodes

Programmable Devices

The source code shown in the main section of the program that calls functions PgmAddr, PgmChk and ChkRsp in Appendix 1 - Source Code details a very simple routine to program the addresses into slave nodes. It programs 15 slave nodes sequentially starting with address 1 (0001) and finishing with address 15 (1111). To ensure the response from the slave node whose address is being set is captured, the MC68HC55's clock period is set to SCLK divided by 4 and the interframe delay is set to 32 bit times. This is achieved by writing $B0 to the MC68HC55's DSIxCTRL register (refer to sub section: 'Initialisation of the MC68HC55 SPI Peripheral's Registers').

Pre-programmed Devices

When the network is configured with pre-programmed devices the initialisation procedure is similar to that of programmable devices. An initialisation command that contains the address of the slave node being initialised is sent to that slave node. The slave node then responds to the initialisation command to let the master know it is present in the network.

Summary

This Application Note has discussed a total system solution using a full suite of Freescale ICs. Although the initial target application is automotive airbag systems it could be used in other applications which require remote sensors. The Distributed Systems Interface (DSI) has many advantages in that it allows flexibility of system design, is easily expandible and allows the central module size to decrease while the system content grows.

References

1. DSI Specification; internal Freescale document
2. MC68HC55 SPI Peripheral Specification; data sheet, order number MC68HC55/D
3. BEM Specification; only available with DSI Evaluation System (contact sales office for more details)
4. MC33790 Physical Layer ASIC Specification; internal Freescale document
5. MC68HC912B32 Advance Information; order number MC68HC912B32/D

Appendix 1 - Source Code

```
/*********************************************************************************
FILE :  B32DSI.h

Header file referenced in program InitB32DSI.c to define the register addresses of the
COP, PWM and SPI

*********************************************************************************/

/* Define COP register */
                     #define COPBASE (volatile char *const)(0x16)
                     #define COPCTL (*(COPBASE+0))


/* Define PORTA for general purpose I/O */
                     #define PTABASE (volatile char *const) (0x00)
                     #define PORTA (*(PTABASE+0))
                     #define DDRA (*(PTABASE+2))


/* Define PWM registers */
                     #define PWMBASE (volatile char *const)(0x40)
                     #define PWCLK (*(PWMBASE+0))
                     #define PWPOL (*(PWMBASE+1))
                     #define PWEN (*(PWMBASE+2))
                     #define PWRES (*(PWMBASE+3))
                     #define PWPER0 (*(PWMBASE+0xc))
                     #define PWDTY0 (*(PWMBASE+0x10))
                     #define PWCTL (*(PWMBASE+0x14))
                     #define PORTPP (*(PWMBASE+0x16))
                     #define DDRP (*(PWMBASE+0x17))


/* Define SPI Registers */
                     #define SPIBASE (volatile char *const)(0xd0)
                     #define SP0CR1 (*(SPIBASE+0))
                     #define SP0CR2 (*(SPIBASE+1))
                     #define SP0BR (*(SPIBASE+2))
                     #define SP0SR (*(SPIBASE+3))
                     #define SP0DR (*(SPIBASE+5))
                     #define PORTS (*(SPIBASE+6))
                     #define DDRS (*(SPIBASE+7))
                     #define PURDS (*(SPIBASE+0xb))


/*********************************************************************************
FILE : InitB32DSI.c

C code to be programmed into the FLASH of the HC912B32 which initialises the PWM and
SPI  of the HC912B32 then sets up the DSI registers on the MC68HC55 SPI Peripheral
I.C. before programming the address into 15 programmable slave nodes
```

```
This source code is example code that could be used to perform initialisation.

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 licensed under its patent rights nor the
right of others.  Freescale products are not designed, intended or authorised for use
as components intended for surgical implant into the body, or other applications
intended to support life, or any other application in which 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 intended or unauthorised
application, Buyer shall indemnify and hold Freescale and its officers, employees,
subsidiaries, affiliates, and distributors harmless against all claims, costs,
damages, expenses and reasonable attorney fees arising out of, directly or indirectly,
any claim of personal injury or death associated with such unintended or unauthorised
use, even if such a claim alleges that Freescale was negligent regarding the  design
or manufacture of the part.  Freescale and the Freescale logo are registered trademarks
of Freescale Semiconductor, Inc.

**********************************************************************************/

                     #include <stdio.h>
                     #include “B32DSI.h”
                     #define ArraySize 0x0F  /* (no. of slave nodes) */

                     short TBytes [ArraySize];  /* Array of bytes to be transmitted */
                     short RBytes [ArraySize];  /* Array of received bytes */
                     short ChkBytes [ArraySize];  /* Array of check bytes to check
                     received data is correct */
                     short ErrorCode [ArraySize];  /* Array of Error codes */


/* This function sets up the PWM of the HC912B32 */
                     void InitPWM(void)
                     {
                        PWCLK = 0x00;  /* Don’t divide A clock */
                        PWPOL = 0x00;  /* Use A clock and polarity is low until duty
                     count is reached */
                        PWPER0 = 0x1b; /* Period of 3.5µs (frequency of 285.71 kHz) */
                        PWDTY0 = 0x0d; /* Duty cycle is 50% */
                        DDRP = 0x01;   /* Set up PTP0 as output */
                        PWCTL = 0x00;  /* Left aligned mode as CENTR=0 */
                        PWEN = 0x01;   /* Enable the PWM */
                     }

/* This function sets up the SPI of the HC912B32 */
                     void InitSPI(void)
                     {
                        PURDS = 0x00;  /* Leave as normal conditions */
                        SP0BR = 0x00;  /* SPI clock frequency is 4 MHz */
                        SP0CR1 = 0x00; /* CPOL = CPHA = 0 */
                        DDRS = 0xe0;   /* Set up PTS[7:5] as outputs (SS, SCLK and
                     MOSI) */
```

```
                        SP0CR1 = 0x10; /* Enable master mode and SSOE=0, therefore SS
                     is GP I/O */
                        SP0CR1 = 0x50; /* Enable SPI system now */
                     }

/* This function is called by the SpiBurst function.  It writes information into the
SPI data register and when the SPI transfer complete flag is set it sends back the
information received by the SPI */
                     short TransmitReceive(short SendByte)
                     {
                        int dummy=0;
                        short result;

                        SP0DR = SendByte;
                        while ((SP0SR & 0x80) == 0)   /* Wait until SPI transfer
                     complete flag is set */
                        {
                           dummy++;
                        }
                        result = SP0DR;
                        return (result);
                     }

/* This function sets up the SPI Burst transfer routine. It enables the MC68HC55 SPI
Peripheral by driving CS low and then transfers the required no. of bytes of data to
complete the burst transfer. It finishes by driving CS high and disabling the MC68HC55
SPI Peripheral */
                     void SpiBurst(int ByteCount)
                     {
                        int count;

                        /* CS to go LOW */
                        PORTS = 0x00;

                        /* Transmit bytes */
                        for (count=0; count<ByteCount; count++)
                        {
                           RBytes[count] = TransmitReceive(TBytes[count]);
                        }

                        /* CS to go HIGH */
                        PORTS = 0x80;
                     }

/* This function sets up the DSI registers */
                     int SetupDSI(void)
                     {
                        int ErrCnt=0;
                        int ChkCnt;

                         /* Set up DSI/D Registers */
                        TBytes[0] = 0x85;             /* Send 85 (write cmd to reg.
                     addr. 5) */
```

```
                          TBytes[1] = 0xb0;              /* Rec./Transmit Interrupts
                       disabled, Msg size 20Bits (16 + 4CRC) */
                          TBytes[2] = 0x00;             /* Set up ch. 1 ctrl reg to be
                       the same */
                          TBytes[3] = 0x01;             /* Only ENABLE ch. 0 (0x03 to
                       enable ch. 0 & 1, 0x02 to enable ch.1 only) */

                          SpiBurst(4);                  /* Transmits the info. set up
                       by TBytes to the DSI/D */

                          TBytes[0] = 0x05;            /* Cmd. to read DSI/D registers
                       starting address 5 */
                          SpiBurst(4);                       /* RBytes[x] will return
                       the contents of DSI/D registers 5 to 7 */

                          ChkBytes[1] = 0xb0;            /* should match RBytes[1]
                       contents of DSI0CTRL reg. */
                          ChkBytes[2] = 0x00;            /* should match RBytes[2]
                       contents of DSI1CTRL reg. */
                          ChkBytes[3] = 0x01;            /* should match RBytes[3]
                       contents of DSIENABL reg. */

/* Compare RBytes[1 to 3] to ChkBytes[1 to 3] if not equal ErrCnt is incremented &
returned to main */
                          for (ChkCnt=1; ChkCnt<4; ChkCnt ++)
                             if ((RBytes[ChkCnt] & 0x00ff) != (ChkBytes[ChkCnt] &
                       0x00ff)) ErrCnt++;

                          return (ErrCnt);
                       }

/* This function is here to allow visibility of failures when using debugger tool */
                       void FlagError(int Err, int Adr)
                       {
                          ErrorCode[Err]=Adr;
                       }

/* This function checks that the TFNF0 flag is set */
                       void TFFlag(int Address)
                       {
                          TBytes[0] = 0x04;
                          SpiBurst(2);
                          if (!(RBytes[1] & 0x02)) {
                             FlagError(2,Address);
                             for (;;);  /* error - loop until device is reset */
                             }
                       }

/* This function waits in a loop until the RFNE0 flag is set */
/* & therefore data has been written into the DSI data registers */
                       void RFFlag(void)
                       {
                          TBytes[0] = 0x04;
```

```
                      SpiBurst(2);
                      while ((RBytes[1] & 0x01) == 0)
                      {
                         SpiBurst(2);
                      }
                   }

/* This function programs the address into the first slave node (no response is
expected ) */
                   void PgmAddr(int Addr)
                   {
                      TFFlag(Addr);         /* Check TFNF0 flag is set */
                      TBytes[0] = 0x80;     /* send 80 (write cmd to DSI0H) receive
                   00 */
                      TBytes[1] = Addr;    /* Pgm Addr cmd into DSI0H then TBytes[2]
                   is DSI0L */
                      TBytes[2] = 0x00;
                      SpiBurst(3);
                      RFFlag();
                   }

/* This function programs the addresses into slave nodes 2 through to 15 and checks
the responses from slave nodes 1 to 14 */
                   void PgmChk(int Addr, int LastAddr)
                   {
                      unsigned short DsiDat;
                      unsigned short DsiChk;
                      int dummy=0;

                      TFFlag(Addr);         /* Check TFNF0 flag is set */
                      TBytes[0] = 0x80;     /* send 80 (write cmd to DSI0H) */
                      TBytes[1] = Addr;     /* Pgm Addr cmd written into DSI0H &
                   DSI0L */
                      TBytes[2] = 0x00;
                      SpiBurst(3);
                      RFFlag();
                      /* CRC Check */
                      if (RBytes[1] & 0x08) {
                         FlagError(4,LastAddr);
                         for (;;);  /* error - loop until device is reset */
                         }
                      TBytes[0] = 0x00;     /* Read DSI0H and DSI0L */
                      SpiBurst(3);
                      DsiDat = RBytes[1] | 0x0000;
                      /* Now shift DsiDat 8 times */
                      DsiDat <<= 8;
                      DsiDat = DsiDat+RBytes[2];

                   /* Now need to check the response ie that bits[15:12] = Prev.
                   Slave Addr */
                      DsiDat >>= 12;
                      DsiChk = LastAddr | 0x0000;
                      if (DsiDat != DsiChk) {
                         FlagError(6,LastAddr);
                         for (;;);  /* error - loop until device is reset  */
                         }

/* This function checks the response to the program address command of the last slave
node (number 15) */
                    void ChkRsp(int Addr, int ADCcmd))
                    {
                       unsigned short DsiDat;
                       unsigned short DsiChk;
                       int dummy=0;

                       TFFlag(Addr);           /* Check TFNF0 flag is set */
                       TBytes[0] = 0x80;    /* send 80 (write cmd to DSI0H) */
                       TBytes[1] = 0x00;    /* cmd written into DSI0H & DSI0L */
                      TBytes[2] = 0xADCcmd;   /*dummy command to allow response from
                    final pgm addr cmd to be captured */
                       SpiBurst(3);
                       RFFlag();
                       /* CRC Check */
                       if (RBytes[1] & 0x08) {
                          FlagError(4, Addr);
                          for (;;);  /*error - loop until device is reset */
                          }
                       TBytes[0] = 0x00;     /* Read DSI0H and DSI0L */
                       SpiBurst(3);
                       DsiDat = RBytes[1] | 0x0000;
                       /* Now shift DsiDat 8 times */
                       DsiDat <<= 8;
                       DsiDat = DsiDat+RBytes[2];

                    /* Now need to check the response ie that bits[15:12] = Slave
                    Addr */
                       DsiDat >>= 12;
                       DsiChk = Addr | 0x0000;
                       if (DsiDat != DsiChk) {
                           FlagError(6,Addr);
                           for (;;);  /* error - loop until device is reset */
                           }

                    int main (int argc, char* argv[] )
                    {
                       int Result=-1;
                       short SlaveNum;  /* the address of the slave node to be
                    programmed */
                       short PrevAddr;    /* the address of the previous slave node
                    whose response requires to be checked */

                       COPCTL = 0x00;  /* Disable COP resets */
                       InitPWM();   /* Set up PWM */
                       InitSPI();   /* Set up SPI */

                       Result = SetupDSI();      /* Set up DSI ctrl and enable
                    registers (Result=0) if all OK*/
                       if (Result != 0) {
                          FlagError(1,1); /* need to stop program as well */
                          for (;;); /* error - loop until device is reset */
                          }

  /* Program the address into each slave node and check response */
  /* For this version no. of slave nodes is 15  (the maximum) */

                       PgmAddr(0x01);
                      for (SlaveNum=0x02, PrevAddr=0x01; SlaveNum<0x10; SlaveNum++,
                    PrevAddr++)
                       {
                          PgmChk(SlaveNum, PrevAddr);
                       }
                       ChkRsp(0x0F, 0x02);

                       return;
                    }
```