HOLT INC. INTEGRATED CIRCUITS

December 2017

AN-140

HI-3585 Design Example Applications Note

INTRODUCTION

The Holt HI-3585 is a single-chip ARINC 429 Serial Peripheral Interface (SPI) IC incorporating a single ARINC 429 receive channel and transmitter. Analog line drivers and receivers are included to allow the user to connect directly to an ARINC bus. The HI-3585 interfaces to a host CPU via the 4-wire SPI. Op codes are used to control the flow of information between the host and the HI-3585. Automatic label recognition, transmit and receiver protocol options and 32-word data buffering FIFOs are included on-chip. A complete description of the HI-3585 can be found at the Holt IC web-site www.holtic.com.

This applications note describes a simple interface design between the HI-3585 and a development board (DEMO9S12XDT512 Rev. C) for the Freescale MC9S12XDT512 microcontroller. The C code routines are shown as examples of how a user may program the HI-3585 by pre-loading and verifying ARINC 429 labels for auto-recognition, transmitting data, polling the status register, and retrieving received ARINC 429 data from the receiver.

HARDWARE DESIGN

An example circuit is shown in Figure 1, which outlines the main connections between the HI-3585 and the DEMO9S12XDT512 (DEMO).

The HI-3585 status and SPI pins are connected to the DEMO as shown in the following table.

MC9S12XDT512	HI-3585
MOSI0	SI
MISO0	SO
SCK0	SCK
$\overline{\text{SS0}}$	$\overline{\text{CS}}$
PA2	MR
PA1	RFLAG
PA0	TFLAG

SOFTWARE DESIGN

All C code was written and compiled with CodeWarrior V4.1 software that accompanied the DEMO board from Freescale. More information about this board may be found at www.freescale.com.

The DEMO board was configured to use SPI0 with the SPI data clocked on the rising edge of SCK and the data changing on the falling edge of SCK. See the Freescale micro controller data sheet (Rev 2.15) for a full description of the SPI control registers.

The SPI control register 1, SPICR1, is set as follows for the design examples in this application note:

SPIE	**(bit 7):**	**SPI interrupts are disabled**
SPE	**(bit 6):**	**SPI enabled, port pins are dedicated to SPI functions**
SPTIE	**(bit 5):**	**SPI Transmitter empty flag disabled**
MSTR	**(bit 4):**	**SPI is in master mode**
CPOL	**(bit 3):**	**Active-high clocks selected. In idle state SCK is low**
CPHA	**(bit 2):**	**Sampling of odd data occurs at odd edges (1, 3, 5 etc.) of the SCK clock**
SSOE	**(bit 1):**	**Slave select output is enable**
LSBFE	**(bit 0):**	**Data is transferred most significant bit first**

(For a full description of SPICR1, please see 3585Ademo.c, attached and MC9S12XDT512.h, V2.02, which is available in the Freescale DEMO package.)

The following short subroutines were written for the examples in order to simplify instructions. The routines enable SPI data transfers between the DEMO board and the HI-3585.

txrx8bits(x,x):	**transfers 8 bit read/write data**
txOpCode(x,x):	**transmits 8 bit data**

A more detailed explanation of these subroutines may be found in the attached example program, 3585Ademo.c.

An expanded header file, 3585Ademo.h, written to accompany the C program, is also attached. The header contains definitions for status register bits, control register bits, LEDs and switches.

The source code for both the C file (3585Ademo.c) and header file (3585Ademo.h) listed in this applications note are also available in electronic format by contacting the Holt Sales Department at sales@holtic.com.

BASIC OPERATION

Control Word

The control word is a 16-bit register used to configure the device. The control word register bits, CR15-CR0, are loaded from SPI with opcode 10hex.

The following code loads the control word.

```
// clear SPI status register
dummy = SPI0SR;
// writing opcode to Data Reg starts SPI xfer
SPI0DR = 0x10;
while (!SPI0SR_SPIF) {;}
// read Rx data in Data Reg to reset SPIF
dummy = SPI0DR;

// write upper Control Register
SPI0DR = (char)(ControlReg >> 8);
while (!SPI0SR_SPIF) {;}
dummy = SPI0DR;
```

```
// write lower Control Register
SPI0DR = (char)(ControlReg & 0xFF);
while (!SPI0SR_SPIF) {;}
dummy = SPI0DR;
```

Load Transmitter FIFO

The transmitter FIFO is loaded with one 32-bit word at a time with op code 0Ehex. The most significant bit of data must follow the last op code bit.

The following code loads one word in the transmitter FIFO:

```
// send op code (ignore returned data byte)
dummy = txrx8bits(0x0E,1);
// send MS byte (ignore returned data byte)
dummy = txrx8bits((char)(TxBusWord[i]>>24),1);
// send next byte (ignore returned data byte)
dummy =txrx8bits((char)((TxBusWord[i]>>16)&0xFF),1);
// send next byte (ignore returned data byte)
dummy =txrx8bits((char)((TxBusWord[i]>>8)& 0xFF),1);
// send LS byte (ignore returned data byte)
dummy = txrx8bits((char)(TxBusWord[i] & 0xFF),1);
```

Enable Transmitting TX FIFO

If control word bit 14 = 0, TFLAG will be low when the TX FIFO contains at least one word. In 3585Ademo.c, data transmission begins after four words are loaded.

There are two ways to transmit data with the HI-3585. If control word bit 13 = 1, then transmission begins as soon as data is available in the TX FIFO. If control word bit 13 = 0, op code 12hex must be written to HI-3585 in order to enable transmission.

The following code demonstrates how the single op code is necessary to initiate transmission.

```
// enable ARINC bus transmit, return only after
// completion
txOpCode (0x12,1);
```

Unload RX FIFO

If control word bit 15 = 0, RFLAG will be low when the RX FIFO contains at least one valid ARINC word. To retrieve the data there are two op codes available. Op code 08hex will retrieve one word at a time. Op code 09hex will retrieve all words in the RX FIFO.

The following code uses op code 08hex to retrieve one word and to transfer that word, 8 bits at a time, into the variable rxdata.

```
 // send op code (ignore returned data byte)
 rxdata = txrx8bits(0x08,1);

// send dummy data
// receive and left-justify most signif. byte

j = txrx8bits(0x00,1);
rxdata = (j << 24);

// send dummy data
// receive, left-shift then OR next byte
j = txrx8bits(0x00,1);
rxdata = rxdata | (j << 16);

// send dummy data
// receive, left-shift then OR next byte
j = txrx8bits(0x00,1);
rxdata = rxdata | (j << 8);

// send dummy data
// receive and OR the least signif. byte
j = txrx8bits(0x00,1);
rxdata = rxdata | j ;
```

LABELS

The HI-3585 has user-programmable label recognition for any combination of the 256 possible labels. There are three options to program all labels. They can all be set (op code (03hex), reset (op code 02hex), or all 256 may be programed in any combination (op code 06hex). To verify the 256 labels are set/reset op code 0Dhex is used to read back all 256 labels. The labels will not be reset after having been read.

The following code examples show how each op code is used and how program arrays can keep track of the label memory.

Reset all HI-3585 ARINC label selections

```
// reset all label bits in HI-358x device
txOpCode(02,1);
// reset all bits all 32-bytes of global LabelArray[]
for (i=0; i<32; i++) {
  LabelArray[i] = 0;}
```

Set all HI-3585 ARINC label selections

```
// set all label bits in HI-358x device
txOpCode(03,1);
// set all bits all 32-bytes of global LabelArray[]
for (i=0; i<32; i++) {
  LabelArray[i] = 0xFF;}
```

Copy HI-3585 ARINC label selections from LabelArray[ ] to HI-3585 ie. Write 256 Label memory to HI-3585

```
// send op code (ignore returned data byte)
dummy = txrx8bits(0x06,1);

// send 32 bytes of ARINC label data
for (i=31; i>=0; I--) {
  // send 1 byte of label data, ignore returned data
  //byte
dummy = txrx8bits(LabelArray[i],1);
}
```

Verify match: HI-3585 ARINC label selections to LabelArray[ ] ie. Read 256 Label memory from HI-3585

```
// send op code (ignore returned data byte)
rxbyte = txrx8bits(0x0D,1);

j = 0xffff;
// starting at high end, read 8-bit increments of
// ARINC label data
for (i=31; i>=0; i--) {
// send dummy data, read 1 byte of label data
rxbyte = txrx8bits(0,1);
// check for mismatch
if (rxbyte != LabelArray[i]) {
```

STATUS REGISTER

The HI-3585 has an 8-bit status register which can be polled to determine the status of the receiver FIFO and transmitter FIFO. Op code 0Ahex is used to retrieve the status register bits.

The following code is an example of how to use op code 0A hex and store the contents in a variable to be used throughout the program.

```
  // send op code (ignore returned data byte)
  rxdata = txrx8bits(0x0A,1);
  // send dummy data / receive Status Reg byte
  rxdata = txrx8bits(0x00,1);
```

EXAMPLE PROGRAMS

```
/*******************************************************************************
*
*       Copyright (C) 2007 Holt Integrated Circuits, Inc.
*       All Rights Reserved
*
* Filename:     3585Ademo.h
* Author:
* Revision:     1.0
*
* Description: 9S12XDT512 Demo Board Header File
*
* Notes:        Used in Project 3585A.mcp
*
*******************************************************************************/

/* include peripheral declarations */
#include <mc9s12xdt512.h> //Revision V2.02

/* define value for LED's when on and off */
#define ON 0
#define OFF 1

/* define value for switches when up (not pressed) and down (pressed) */
#define UP 1
#define DOWN 0

/* define LED's */
#define LED1 PORTB_PB4
#define LED2 PORTB_PB5
#define LED3 PORTB_PB6
#define LED4 PORTB_PB7

/* define SW's */
#define SW1  PTP_PTP0
#define SW2  PTP_PTP1
#define SW31 PORTB_PB0
#define SW32 PORTB_PB1
#define SW33 PORTB_PB2
#define SW34 PORTB_PB3

/* define chip select outputs for the 3 SPI ports */
#define SPI0_nSS PTP_PTP3
#define SPI1_nSS PTM_PTM3

/* define ARINC device status flags */
#define TFLAG PORTA_PA0
#define RFLAG PORTA_PA1

#define MR    PORTA_PA2
#define MARK  PORTA_PA3
```

Additional Information

Information on the Freescale demonstration board and microcontroller can be found by searching the device type number at www.Freescale.com.

More information on programming with CodeWarrior can be found at www.Freescale.com/CodeWarrior.

```
/* define Status Register bits */
#define RxFIFO_Empty 0x01
#define RxFIFO_HFull 0x02 // Half-Full, 16 (or more) words
#define RxFIFO_Full  0x04
#define TxFIFO_Empty 0x08
#define TxFIFO_HFull 0x10 // Half-Full, 16 (or more) words
#define TxFIFO_Full  0x20

/* define Control Register bits */

// ACTION IF BIT IS SET    //  ACTION IF BIT IS *NEGATED*
//------------------------ // --------------------------------
#define DIVIDE_ACLK 0x0002 // ARINC TX/RX uses undivided ACLK
#define RXSPEED_LOW 0x0001 // ARINC receive bus speed = high
#define TXSPEED_LOW 0x0400 // ARINC transmit bus speed = high
#define LABELS_ON   0x0004 // ARINC word label decoding is off
#define LBL_NOREV   0x0800 // Tx/Rx label bit order is reversed
#define RXPARITY_ON 0x0010 // receive parity off (all 32 bits = data)
#define TXPARITY_ON 0x0008 // transmit parity off (all 32 bits = data)
#define TXPAR_EVEN  0x0200 // transmit parity = odd
#define LOOPBAK_OFF 0x0020 // Tx-to-Rx loop-back is enabled
#define BUSDRV_OFF  0x1000 // ARINC bus transmit is enabled
#define TXAUTOSTART 0x2000 // Tx starts with op code 0x12
#define TXFULL_FLAG 0x4000 // TFLAG output = Tx FIFO Empty
#define RXFULL_FLAG 0x8000 // RFLAG output = Rx FIFO Empty
#define RXDECODE_ON 0x0040 // decoding of Rx ARINC bits 10:9 is off
// next 2 apply only
// if RXDECODE_ON =1
#define RXBIT10_HI  0x0080 // Rx ARINC bit 10 must be low
#define RXBIT9_HI   0x0100 // Rx ARINC bit 9 must be low
```

```
/* define Control Register bits */

// ACTION IF BIT IS SET    //  ACTION IF BIT IS *NEGATED*
//------------------------ // --------------------------------
#define DIVIDE_ACLK 0x0002 // ARINC TX/RX uses undivided ACLK
#define RXSPEED_LOW 0x0001 // ARINC receive bus speed = high
#define TXSPEED_LOW 0x0400 // ARINC transmit bus speed = high
#define LABELS_ON   0x0004 // ARINC word label decoding is off
#define LBL_NOREV   0x0800 // Tx/Rx label bit order is reversed
#define RXPARITY_ON 0x0010 // receive parity off (all 32 bits = data)
#define TXPARITY_ON 0x0008 // transmit parity off (all 32 bits = data)
#define TXPAR_EVEN  0x0200 // transmit parity = odd
#define LOOPBAK_OFF 0x0020 // Tx-to-Rx loop-back is enabled
#define BUSDRV_OFF  0x1000 // ARINC bus transmit is enabled
#define TXAUTOSTART 0x2000 // Tx starts with op code 0x12
#define TXFULL_FLAG 0x4000 // TFLAG output = Tx FIFO Empty
#define RXFULL_FLAG 0x8000 // RFLAG output = Rx FIFO Empty
#define RXDECODE_ON 0x0040 // decoding of Rx ARINC bits 10:9 is off
// next 2 apply only
// if RXDECODE_ON =1
#define RXBIT10_HI  0x0080 // Rx ARINC bit 10 must be low
#define RXBIT9_HI   0x0100 // Rx ARINC bit 9 must be low
```

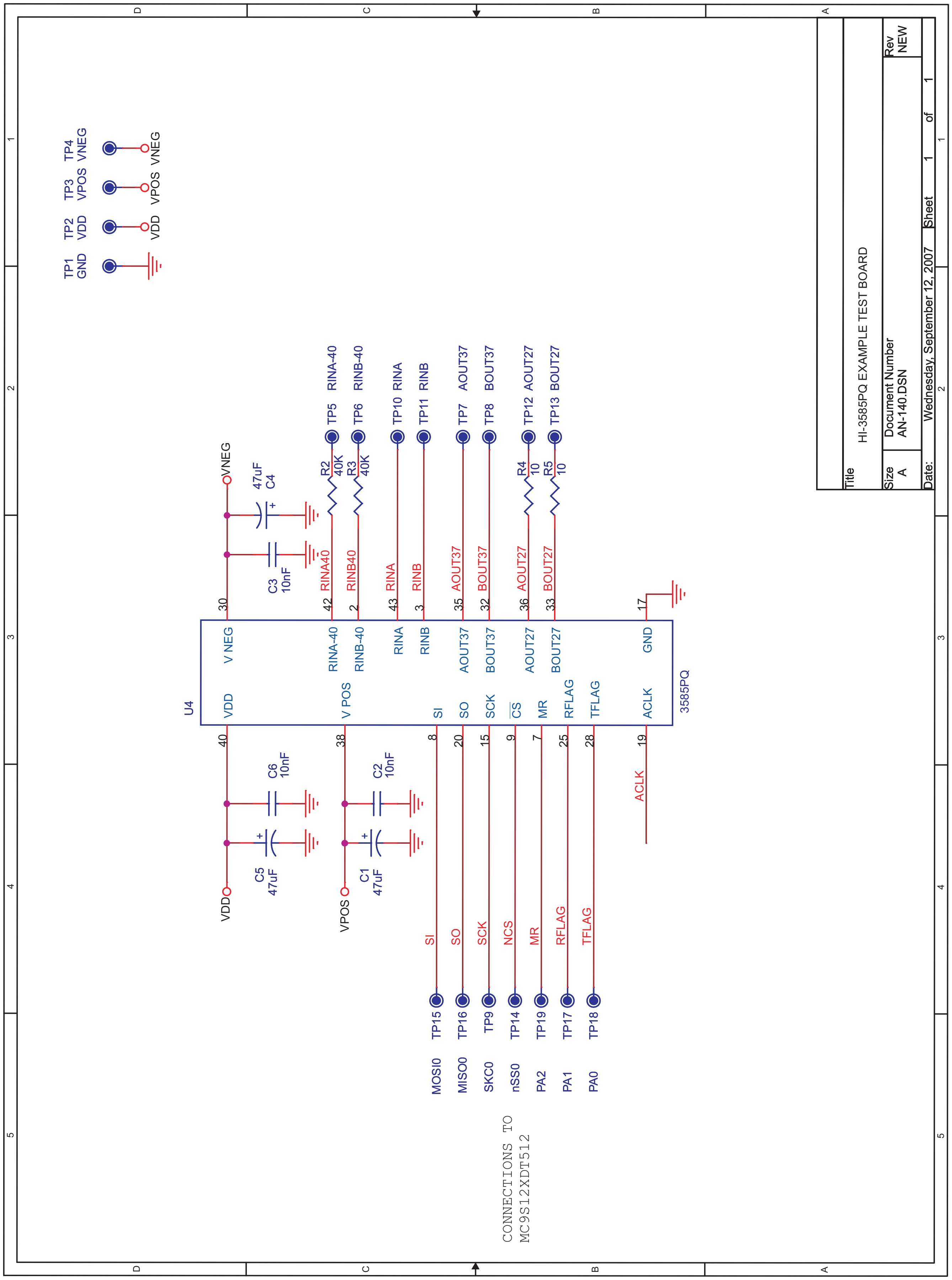
```
/******************************************************************************
*
*       Copyright (C) 2010 Holt Integrated Circuits, Inc.
*       All Rights Reserved
*
* Filename:     3585Ademo.c
* Revision:     1.1
*
* Description: Test Routines for Holt HI-3585, -3587 & -3588
*               ARINC 429 ICs with SPI Interface
*
* Notes:        Uses the Freescale 9S12XDT512 demo board Rev C.
*               Created using CodeWarrior v4.1 for S12X.
*
******************************************************************************/

#include <hidef.h>            // common defines and macros
#include <mc9s12xdt512.h>     // derivative information Rev V2.02
#include <string.h>
#include "xgate.h"
#include "3585Ademo_1.0.h"    // Include the demo board declarations

#pragma LINK_INFO DERIVATIVE "mc9s12xdt512"

#define SOFTWARETRIGGER0_VEC  0x72 /* vector address= 2 * channel id */
#define ROUTE_INTERRUPT(vec_adr, cfdata)                  \
  INT_CFADDR= (vec_adr) & 0xF0;                           \
  INT_CFDATA_ARR[((vec_adr) & 0x0F) >> 1]= (cfdata)

/* Global Variables */

unsigned long RxBusWord[32], TxBusWord[32];

/* Initialize HI-358x Control Register: undivided ACLK, label matching enabled,
   no Rx parity, no Tx parity, no loop-back,
   TFLAG output = Tx FIFO empty, RFLAG output = Rx FIFO empty,
   disable ARINC RxWord bit 10:9 decoding */
unsigned short ControlReg = LABELS_ON;


//====================================================================
/* Initial selections for active ARINC word labels
   Each byte defines 8 label addresses, LS bit =  lowest addr 0 of 7
                                        MS bit = highest addr 7 of 7
     [n]   = array pointer value,
   xxx-yyy = corresponding ARINC label address range

   Example:
   [3] denotes LabelArray[3] which controls label addr 24-31 decimal.
   Bit 0 controls label addr 24, Bit 1 controls label addr 25, etc. */

unsigned char LabelArray[32] = {

// ----------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
// 000-007 008-015 016-023 024-031 032-039 040-047 048-055 056-063
     0x0B,   0x01,   0x01,   0x00,   0x01,   0x00,   0x00,   0x00,
// ----------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
// 064-071 072-079 080-087 088-095 096-103 104-111 112-119 120-127
     0x01,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,
// ----------------------------------------------------------------
//   [16]    [17]    [18]    [19]    [20]    [21]    [22]    [23]
// 128-135 136-143 144-151 152-159 160-167 168-175 176-183 184-191
     0x01,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,
// ----------------------------------------------------------------
//   [24]    [25]    [26]    [27]    [28]    [29]    [30]    [31]
// 192-119 200-207 208-215 216-223 224-231 232-239 240-247 248-255
     0x00,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,   0x00,
// ----------------------------------------------------------------
// label addresses 0,1,3,8,16,32,64 and 128 are enabled
};

 // --------------------------------------------------------------------------
// XGate Setup Routine
// ---------------------------------------------------------------------------
static void SetupXGATE(void) {
  /* initialize the XGATE vector block and
     set the XGVBR register to its start address */
  XGVBR= (unsigned int)(void*__far)(XGATE_VectorTable - XGATE_VECTOR_OFFSET);

  /* switch software trigger 0 interrupt to XGATE */
  ROUTE_INTERRUPT(SOFTWARETRIGGER0_VEC, 0x81); /* RQST=1 and PRIO=1 */

  /* enable XGATE mode and interrupts */
  XGMCTL= 0xFBC1; /* XGE | XGFRZ | XGIE */

  /* force execution of software trigger 0 handler */
  XGSWT= 0x0101;
}


/* ------------------------------------------------------------------
/  Initialization Function. Is called by main() after reset
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  nothing

     Action:  initializes processor configuration registers */

void PeriphInit(void)
{

    DDRB_DDRB0 = 0;           // Port B[0..3] input (SW3 1-4)
    DDRB_DDRB1 = 0;
    DDRB_DDRB2 = 0;
    DDRB_DDRB3 = 0;
    DDRB_DDRB4 = 1;           // Port B[4..7] output (LED1-LED4)
    DDRB_DDRB5 = 1;
    DDRB_DDRB6 = 1;
    DDRB_DDRB7 = 1;

    LED1 = OFF;                                     // Turn Off LEDs
    LED2 = OFF;
    LED3 = OFF;
    LED4 = OFF;

    PUCR_PUPBE = 1;           // Turn on the pullups for SW3 (1-4)

    DDRA_DDRA0 = 0;           // Port A[0..1] input (TFLAG & RFLAG))
    DDRA_DDRA1 = 0;
    PUCR_PUPAE = 1;           // Turn on pullups for TFLAG & RFLAG

    DDRA_DDRA2 = 1;           // Port A[2] output (MR)
    MR = 0;
    DDRA_DDRA3 = 1;           // Port A[3] output (MARK)
    MARK = 0;

    /* ----------------------------------------
    PLL init for 4MHz osc ---> 80MHz bus (max freq)
    load synth register = 9 as (2(9+1)) x 4MHz = 80MHz  */
    SYNR = 10;
    // wait for PLL LOCK flag to go high
    while (!(CRGFLG & 0x08)) {
      ;
    }
    // set PLLSEL bit in Clock Select register
    CLKSEL = 0x80;

    /* ----------------------------------------------------------------
    The Freescale DEMO9S112XDT152 Demo Board has 3 SPI ports.
    Each has pins muxed with other features or general purpose I/O.
    Some SPI port assignments are nonstandard since the board uses
    the 80-QFP package with reduced pin count.

    ----------------------------------------------------------------
    SPI0 - All 4 pins are available. For the 80-QFP package, SPI0
    only appears on Port M [5:2]. The Module Routing Register MODRR
    bit 4 must be set to make the SPI appear on Port M. The SPI0
    Control Register SPE enable bit must be set.
                  Signal    U1 Pin       J1
                   Name     80PQFP      Pin
                  ------    ------     ------
                   SCK0       70         21
                   /SS0       72         23
                  MOSI0       71         17
                  MISO0       73         19

    The next 5 lines ONLY apply if using SPI0...    */
    DDRM_DDRM3 = 1;       // make Port M[3] output (SPI0 /SS)
    SPI0_nSS = 1;         // default SPI0 /SS state = high
    MODRR_MODRR4 = 1;     // Route SPI0 to Port M[5:2] by setting MODRR[4]
    DDRP_DDRP0 = 0;          // Make Port PP_0 input (pushbutton SW1)
    DDRP_DDRP1 = 0;       // Make Port PP_1 input (pushbutton SW2)

    /*-----------------------------------------------------------------
    SPI1 - All pins are available, appearing on Port P[3:0] when SPI1
    Control Register SPE enable bit is set. Two SPI1 I/O pins are used
    for Freescale demo board pushbuttons SW1 and SW2. If using SPI1,
    SW1 and SW2 should be disconnected from the microprocessor pins, and
    could be be reassigned to spare I/O if needed. Remove "User Jumpers"
    1 & 2 to disconnect pushbuttons. Suggestion: Edit the header file to
    comment-out SW1 and SW2 symbols, so the complier doesn't accidentally
    create conflicts if SW1 or SW2 references persist in C program.
                  Signal    U1 Pin       J1
                   Name     80PQFP      Pin
                  ------    ------     ------
                   SCK1        2         30
                   /SS1        1         32
                  MOSI1        3         11
                  MISO1        4          9

    The next 2 lines ONLY apply if using SPI1. ..   */
    //DDRP_DDRP3 = 1;     // make Port P[3] output (SPI1 /SS)
    //SPI1_nSS = 1;       // default SPI1 /SS state = high

    /*-----------------------------------------------------------------
    SPI2 - When using the 80-pin PQFP, 3 of 4 pins are available on
    Port P bits 4,5 & 7 when SPI2 Control Register SPE bit is asserted..
                  Signal    U1 Pin       J1
                   Name     80PQFP      Pin
                  ------    ------     ------
                   SCK2       78          6
                   /SS2       --         --   not avail in 80-PQFP
                  MOSI2       79         36
                  MISO2       80         34

    NOTE: When using the 80-QFP package, SPI2 lacks an /SS output. The
    /SS output is not essential in many SPI applications, but is needed
    for HI-3585 interface. A bit-banged general purpose output can provide
    the /SS function BUT this will require modifications to the 8-bit
    SPI transfer functions that use hardware "auto /SS" control.

    (no SPI2 initialization provided due to the above compatibility issue)

    -------------------------------------------------------------------

    If changing from SPI0 to SPI1, from this point on find/replace all
    instances of "SPI0" with "SPI1" (and comment/uncomment init code blocks
    above) */

    SPI0CR1 = SPI0CR1_SPE_MASK|SPI0CR1_SSOE_MASK;
    // enable SPI0 mode fault
    SPI0CR2 = SPI0CR2_MODFEN_MASK;
    // set baud rate at 5.0 Mbps  (80MHz / 8)
    SPI0BR = 0x02;
    // read status (the write has no effect)
    SPI0SR = SPI0SR;
    // toggle MSTR bit to idle SPI0 in Master State
    SPI0CR1 = SPI0CR1_SPE_MASK|SPI0CR1_SSOE_MASK|SPI0CR1_MSTR_MASK;

}   /* PeriphInit() */
```

```
/* -----------------------------------------------------------------
/  Init Processor TxArray with Test Data Function
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  nothing

     Action:  loads the 32-element x 32-bit global array
              TxBusWord[i] with this arbitrary test data:

              TxBusWord[0]  = 0x01010101
              TxBusWord[1]  = 0x02020202
              TxBusWord[2]  = 0x03030303
                       (. . . . )
              TxBusWord[8]  = 0x09090909
              TxBusWord[9]  = 0x10101010
              TxBusWord[10] = 0x11111111
                       (. . . . )
              TxBusWord[29] = 0x30303030
              TxBusWord[30] = 0x31313131
              TxBusWord[31] = 0x32323232

Example call: InitTxArray(); //        */

void InitTxArray (void) {

  char i;
  unsigned long j = 0;

  for (i=0; i<32; i++) {
    j++;
    if ((j&0xF) == 0xA) j = j+6;
    TxBusWord[i] = (j<<24)+(j<<16)+(j<<8)+j;
  }

}  /* InitTxArray() */

/* -----------------------------------------------------------------
/  Init Processor TxArray with Test Data Function
/  -----------------------------------------------------------------

Argument(s):  rolling One or rolling Zero

     Return:  nothing

     Action:  loads the 32-element x 32-bit global array
              TxBusWord[i] with


Example call: InitTxArrayRolling(1); //        */

void InitTxArrayRolling (unsigned char x) {

  char i;
  unsigned long j = 0x80000000;

  for (i=0; i<32; i++) {
    TxBusWord[i] = j >> i;
  }

  if (x == 0x0){
    for (i=0; i<32; i++){
      TxBusWord[i] = ~TxBusWord[i];
    }
  }

}  /* InitTxArrayRolling() */


/* -----------------------------------------------------------------
/  Clear Processor TxArray Function
/  -----------------------------------------------------------------
```

```
Argument(s):  none

     Return:  nothing

     Action:  clears the 32-element x 32-bit global array
              TxBusWord[0] through TxBusWord[31] = 0

Example call: ClearTxArray();   //        */

void ClearTxArray (void) {

  char i;

  for (i=0; i<32; i++) {
    TxBusWord[i] = 0;
  }

}  /* InitTxArray() */


/* ------------------------------------------------------------------
/  Send 8-Bit Op Code without Data
/  ------------------------------------------------------------------

Argument(s):  txbyte, return_when_done

     Return:  nothing

     Action:  This function sends 8 bits using SPI0. This function
              can only be used with the following HI-358x op codes
              that DO NOT have an associated Data Field:

              OP CODE   HI-358x ACTION
               0x01     Master Reset
               0x02     Reset/disable all label selections
               0x03     Set/enable all label selections
               0x11     Reset the Transmit FIFO
               0x12     Enable ARINC bus transmission

              If return_when_done is True, the function waits for transfer
              completion before returning, which may be needed for back-to-
              back commands. If return_when_done is False, the function
              returns immediately after initiating the transfer.

Example Use:  tx8bits(0x01,0); // apply MR, return immediately */

void txOpCode (unsigned char txbyte, unsigned char return_when_done) {

  unsigned char dummy;

  dummy = SPI0SR;          // clear SPI status register

  SPI0DR = txbyte;         // write Data Register to begin transfer

  if (return_when_done) {  // optional wait for SPIF flag
    while (!SPI0SR_SPIF) {
      ;
    }
  }
  dummy = SPI0DR;           // clear the SPIF bit SPI0SR_SPIF

}    /* txOpCode */

/* ------------------------------------------------------------------
/  SPI0 8-Bit Send Data / Receive Data Function
/  ------------------------------------------------------------------

Argument(s):  txbyte, return_when_done

     Return:  rxbyte

     Action:  Using SPI0, this function sends txbyte and returns rxbyte
              as part of a chained multi-byte transfer. The calling
              program controls the /SS chip select instead of using the
              automatic /SS option available in the Freescale SPI port.
              This permits simple chaining of op code commands and Tx/Rx
              data as a series of 8-bit read/writes followed by /SS
              deassertion by the calling program.

              If return_when_done is True, the function waits for transfer
              completion before returning, which may be needed for back-to-
              back commands. If return_when_done is False, the function
              returns immediately after initiating the transfer.

Example Call: rcv_byte = txrx8bits(0xFF,1) // sends data 0xFF then returns
                                           // data when xfer is done  */

unsigned char txrx8bits (unsigned char txbyte, unsigned char return_when_done) {

  unsigned char rxbyte;

  rxbyte = SPI0SR;         // clear SPI status register

  SPI0DR = txbyte;         // write Data Register to begin transfer

  if (return_when_done) {  // optional wait for SPIF flag
    while (!SPI0SR_SPIF);
  }
  // get received data byte from Data Register
  return rxbyte = SPI0DR;

}    /* txrx8bits */

/* ------------------------------------------------------------------
/  Write HI-358x Control Register Function
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  nothing

     Action:  Using SPI0, this function transmits op code 0x10 plus
              16-bits of CR data using three 8-bit SPI transfers.
              The global variable ControlReg is loaded to Control reg.

Example call: ControlReg = 0x1234;  // set value to be loaded
              writeControlReg();    // SPI transfer     */

void writeControlReg (void) {

  unsigned char dummy;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;
  //============================================
  // slower option (commented-out)
  //txrx8bits(0x10,1);                      // op code
  //txrx8bits((char)(ControlReg >> 8),1);   // upper CR
  //txrx8bits((char)(ControlReg & 0xFF),1); // lower CR
  //============================================
  // faster option (ends at "====" line)
  dummy = SPI0SR;       // clear SPI status register
  SPI0DR = 0x10;        // writing opcode to Data Reg starts SPI xfer
  while (!SPI0SR_SPIF) {
    ;
  }
  dummy = SPI0DR;       // read Rx data in Data Reg to reset SPIF
  //--------------------------------------------
  SPI0DR = (char)(ControlReg >> 8); // write upper Control Register
  while (!SPI0SR_SPIF) {
    ;
  }
  dummy = SPI0DR;       // read Rx data in Data Reg to reset SPIF
  //--------------------------------------------
  SPI0DR = (char)(ControlReg & 0xFF); // write lower Control Register
  while (!SPI0SR_SPIF) {
    ;
  }
  dummy = SPI0DR;       // read Rx data in Data Reg to reset SPIF
  //============================================
  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;

} /* writeControlReg() */

/* -----------------------------------------------------------------
/  Read HI-358x Control Register Function
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  16-bit Control Register value

     Action:  Using SPI0, this function transmits op code 0x0B plus
              16-bits of dummy data using three 8-bit SPI transfers.
              The last 2 transfers receive Control Register data bytes
              that are combined to yield the 16-bit value returned */

unsigned short readControlReg (void) {

  unsigned short rxword;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;
  //--------------------------------------------
  rxword = SPI0SR;        // clear SPI status register
  SPI0DR = 0x0B;          // writing opcode to Data Reg starts SPI xfer
  while (!SPI0SR_SPIF);   // wait for SPIF flag assertion
  rxword = SPI0DR;        // read/ignore Rx data in Data Reg, resets SPIF
  //--------------------------------------------
  SPI0DR = 0;             // send dummy data, receive upper Control Reg
  while (!SPI0SR_SPIF);   // wait for SPIF flag assertion
  rxword = (SPI0DR<<8);   // read upper Control Reg byte in Data Reg
  //--------------------------------------------
  SPI0DR = 0;             // send dummy data, receive lower Control Reg
  while (!SPI0SR_SPIF);   // wait for SPIF flag assertion
  rxword = rxword|SPI0DR; // read lower Control Reg byte in Data Reg
  //--------------------------------------------
  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
  //--------------------------------------------
  return rxword;

} /* readControlReg() */

/* -----------------------------------------------------------------
/  Read HI-358x Status Register Function
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  8-bit Status Register data

     Action:  Using SPI0, this function transmits op code 0x0A plus
              1 byte of dummy data using two 8-bit SPI transfers.
              The received byte from SPI transfer #2 is returned  */

unsigned char readStatusReg (void) {

  unsigned char rxdata;
  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;
  // send op code (ignore returned data byte)
  rxdata = txrx8bits(0x0A,1);
  // send dummy data / receive Status Reg byte
  rxdata = txrx8bits(0x00,1);
  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
  return rxdata;

} /* readStatusReg() */


/* ------------------------------------------------------------------
/  Write HI-358x ACLK Divisor Function
/  ------------------------------------------------------------------

Argument(s):  divisor (only decimal 1,2,4,8 or 10 is valid)

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x07 plus
              1 byte Divisor data using two 8-bit SPI transfers.

Example Use:  writeACLKdiv(10); // writes ACLK div-by-10    */

void writeACLKdiv (unsigned char divisor) {

  unsigned char rxdata;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;
  // send op code (ignore returned data byte)
  rxdata = txrx8bits(0x07,1);
  // send 8-bit divisor (ignore returned data byte)
  rxdata = txrx8bits(divisor,1);
  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;

} /* writeACLKdiv() */


/* ------------------------------------------------------------------
/  Read HI-358x ACLK Divisor Function
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  8-bit Status Register data

     Action:  Using SPI0, this function transmits op code 0x0C plus
              1 byte of dummy data using two 8-bit SPI transfers.
              The received byte from SPI transfer #2 is returned */

unsigned char readACLKdiv (void) {

  unsigned char rxdata;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;
  // send op code (ignore returned data byte)
  rxdata = txrx8bits(0x0C,1);
  // send dummy data / receive Status Reg byte
  rxdata = txrx8bits(0x00,1);
  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
  return rxdata;

} /* readACLKdiv() */


/* -----------------------------------------------------------------
/  Read HI-358x next RxFIFO Word Function
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  next 32-bit ARINC word in RxFIFO

     Action:  Using SPI0, this function transmits op code 0x08 plus
              4 bytes of dummy data using five 8-bit SPI transfers.
              The received bytes from SPI transfers #2-5 are merged
              to yield the 32-bit ARINC word that is returned

unsigned long read1RxFIFO (void) {

    unsigned long j, rxdata;  // long = 32 bits

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI1 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI1 /SS strobe
  SPI0_nSS = 0;
  // send op code (ignore returned data byte)
  rxdata = txrx8bits(0x08,1);
  // send dummy data / receive and left-justify most signif. byte
  j = txrx8bits(0x00,1);
  rxdata = ( j << 24);
  // send dummy data / receive, left-shift and OR next byte
  j = txrx8bits(0x00,1);
  rxdata = rxdata | (j << 16);
  // send dummy data / receive, left-shift and OR next byte
  j = txrx8bits(0x00,1);
  rxdata = rxdata | (j << 8);
  // send dummy data / receive and OR the least signif. byte
  j = txrx8bits(0x00,1);
  rxdata = rxdata | j;
  // negate the SPI1 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI1 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
  return rxdata;

} /* read1RxFIFO() */

/* -----------------------------------------------------------------
/  Write HI-358x TxFIFO Function
/  -----------------------------------------------------------------

Argument(s):  words_to_send (number of words to load, range 1 to 32)

     Return:  none
     Action:  Using SPI0, this function transmits op code 0x0E. Then
              one ARINC word is written using four 8-bit transfers.
              The returned data from the 4 transfers is ignored. The
              word count is decremented

Example Use:  writeTxFIFO(11); // writes 11 words to TxFIFO from the
                               // 32-element array TxBusWord[i], sending
                               //  words TxBusWord[0] - TxBusWord[10] */

void writeTxFIFO (unsigned char words_to_send) {

  unsigned char dummy, i;

  for (i=0; i<words_to_send; i++) {

    // disable auto /SS output, reset /SS Output Enable
    SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
    // disable auto /SS output, reset SPI0 Mode Fault
    SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
    // assert the SPI0 /SS strobe
    SPI0_nSS = 0;

    // send op code (ignore returned data byte)
    dummy = txrx8bits(0x0E,1);
    // send MS byte (ignore returned data byte)
    dummy = txrx8bits((char)(TxBusWord[i]>>24),1);
    // send next byte (ignore returned data byte)
    dummy = txrx8bits((char)((TxBusWord[i]>>16) & 0xFF),1);
    // send next byte (ignore returned data byte)
    dummy = txrx8bits((char)((TxBusWord[i]>>8) & 0xFF),1);
    // send LS byte (ignore returned data byte)
    dummy = txrx8bits((char)(TxBusWord[i] & 0xFF),1);

    // negate the SPI0 /SS strobe
    SPI0_nSS = 1;
    // enable auto /SS output, set /SS Output Enable
    SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
    // enable auto /SS output, set SPI0 Mode Fault
    SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
        }
} /* writeTxFIFO() */


/* -----------------------------------------------------------------
/  Reset all HI-358x ARINC label selections
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x02. This
              op code resets all 256 ARINC labels. This function then
              updates the global LabelArray[n] by setting all bits to
              permit status tracking.                              */

void resetAllLabels (void ) {

  unsigned char i;

  // reset all label bits in HI-358x device
  txOpCode(2,1);
  // reset all bits all 32-bytes of global LabelArray[]
  for (i=0; i<32; i++) {
    LabelArray[i] = 0;
  }
} /* resetAllLabels */

/* -----------------------------------------------------------------
/  Set all HI-358x ARINC label selections
/  -----------------------------------------------------------------

Argument(s):  none

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x03. This
```

```
              op code sets all 256 ARINC labels. This function then
              updates the global LabelArray[n] by setting all bits to
              permit status tracking.                              */

void setAllLabels (void ) {

  unsigned char i;

  // set all label bits in HI-358x device
  txOpCode(3,1);
  // set all bits all 32-bytes of global LabelArray[]
  for (i=0; i<32; i++) {
    LabelArray[i] = 0xFF;
  }
} /* setAllLabels */


/* ------------------------------------------------------------------
/  Copy HI-358x ARINC label selections from LabelArray[] to HI-358x
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x06. This
              function then copies the global LabelArray[n] to HI-358x
              label memory so labels are enabled/disabled to match the
              program's LabelArray[].                              */

void copyAllLabels (void ) {

  unsigned char dummy;
  signed char i;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;

  // send op code (ignore returned data byte)
  dummy = txrx8bits(0x06,1);

  // send 32 bytes of ARINC label data
  for (i=31; i>=0; i--) {
  // send 1 byte of label data, ignore returned data byte
    dummy = txrx8bits(LabelArray[i],1);
  }

  // negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
}  /* copyAllLabels */


/* ------------------------------------------------------------------
/  Verify match: HI-358x ARINC label selections to LabelArray[]
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  0xFFFF if LabelArray[n] fully matches HI-358x label memory

              Failing array pointer "n" value 0x0 to 0xFF if a mismatch
              is found. Match testing starts at 0xFF and works toward 0.
              Other mismatches beyond the first failing 8-bit label range
              may exist but are not reported.

     Action:  Using SPI0, this function transmits op code 0x0D. This
              function then compares all 256 bits in HI-358x label
              memory (8 bits at a time) to the corresponding element
              in the program's LabelArray[n]. Return value indicates
              whether or not mismatch is detected.                 */

unsigned short checkAllLabels (void) {

  unsigned char rxbyte;
  signed char i;
  unsigned short j;

  // disable auto /SS output, reset /SS Output Enable
  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;
  // disable auto /SS output, reset SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK;
  // assert the SPI0 /SS strobe
  SPI0_nSS = 0;

  // send op code (ignore returned data byte)
  rxbyte = txrx8bits(0x0D,1);

  j = 0xffff;
  // starting at high end, read 8-bit increments of ARINC label data
  for (i=31; i>=0; i--) {
    // send dummy data, read 1 byte of label data
    rxbyte = txrx8bits(0,1);
    // check for mismatch
    if (rxbyte != LabelArray[i]) {
      // negate the SPI0 /SS strobe
      SPI0_nSS = 1;
      // enable auto /SS output, set /SS Output Enable
      SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
      // enable auto /SS output, set SPI0 Mode Fault
      SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
      // return the failing array pointer value
      return (i);
    }
  }

  // no errors, negate the SPI0 /SS strobe
  SPI0_nSS = 1;
  // enable auto /SS output, set /SS Output Enable
  SPI0CR1 = SPI0CR1 | SPI0CR1_SSOE_MASK;
  // enable auto /SS output, set SPI0 Mode Fault
  SPI0CR2 = SPI0CR2 | SPI0CR2_MODFEN_MASK;
  // return the "no fail" value
  return j;
}  /* checkAllLabels */


// ---------------------------------------------------------------------------
// Main
// ---------------------------------------------------------------------------
void main(void) {

  unsigned short rxword;
  unsigned long i, j, k;
  unsigned char rxbyte;

  EnableInterrupts;
  SetupXGATE();
  PeriphInit();        // initialize microprocessor

  // clear array
  for (i=0; i<32; i++) {
    RxBusWord[i] = 0;
  }

  // apply HI-358x Master Reset.
  // by software, return only after completion...
  // txOpCode (0x01,1);
  // or, by hardware, set MR , delay, reset MR...
  MR = 1;
  for (i = 10; i > 0; i--) ;
  MR = 0;
  // initialize the HI-358x Control Register
  writeControlReg();

  // read back HI-358x Control Register
  rxword = readControlReg();

  // update LED1 to show match or mismatch
  if (rxword == ControlReg) LED1 = ON;
  else LED1 = OFF;

  //write 256 labels
  //labels identified in LabelArray[32]
  copyAllLabels();

  // and read 256 labels
  rxword = checkAllLabels();
  // update LED2 to show label read-back result,
  // match (FFFF) or mismatch (any other value)
        if (rxword == 0xFFFF) LED2 = ON;
        else LED2 = OFF;


  // Load TxData array with 4 words

  TxBusWord[0] = 0x01010101;
  TxBusWord[1] = 0x02020202;
  TxBusWord[2] = 0x03030303;
  TxBusWord[3] = 0x04040404;

  // Load above 4 words to the 3585 TxFIFO.
  // ARINC bus transmit will auto-start if CR13 = 1...
  writeTxFIFO(4);

  rxbyte = readStatusReg();

  // enable ARINC bus transmit, return only after completion
  txOpCode (0x12,1);

  j = 0;

  receiverFIFO:
  // waiting for RFLAG to go low
    while (RFLAG);

  //RFLAG is low, read 1 RX FIFO WORD
  k = read1RxFIFO();

  //compare RXFIFO contents to TXFIFO
  if (k == TxBusWord[j]) LED3 = ON;
  else LED3 = OFF;

  //Labels enable: every 2nd TX word should be received
  j = j + 2;

  goto receiverFIFO;


}//end main(void)
```

Figure 1: Example Circuit



REVISION HISTORY

P/N	Rev	Date	Description of Change
AN-140	A	05/22/09	Update all instances of the product name from HI-3585A to HI-3585
	B	06/18/09	Corrected mistyped part number
	C	07/09/10	Corrected mistyped file name
	D	12/13/17	Update example code