HOLT INC. INTEGRATED CIRCUITS

July 2008

AN-150

HI-3598, HI-3599 Design Example Applications Note

INTRODUCTION

The Holt HI-3598 and HI-3599 are silicon gate CMOS ICs for interfacing eight ARINC 429 receive buses to a high-speed Serial Peripheral Interface (SPI) enabled microcontroller. Either device 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-3598/HI-3599. Automatic label recognition, a 4 word data buffer and on-chip analog line receivers are available on each device. A complete description of the HI-3598 and HI-3599 can be found in the HI-3598 datasheet available at the Holt website at www.holtic.com.

This applications note describes a simple interface design between the HI-3598 or HI-3599 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 Holt devices by pre-loading and verifying ARINC 429 labels for auto-recognition, transmitting internal self test data, polling the status register, and retrieving received ARINC 429 data from each of the 8 receivers.

HARDWARE DESIGN

An example circuit for each device is shown in Figures 1 and 2, which outline the main connections between the HI-3598 or HI-3599 and the DEMO9S12XDT512 board.

The HI-3598/HI-3599 status and SPI pins are connected to the Demo Board as shown in the Table 1.

MC9S12XDT512	HI-3598	HI-3599
MOSI0	SI	SI
MISO0	SO	SO
SCK0	SCK	SCK
$\overline{\text{SS0}}$	$\overline{\text{CS}}$	$\overline{\text{CS}}$
PA2	MR	N/A
PA1	FLAG	FLAG
PA0	FLAG1	N/A
PA4	FLAG2	N/A
PA5	FLAG3	N/A
PA6	FLAG4	N/A
PA7	FLAG5	N/A
PB0	FLAG6	N/A
PB1	FLAG7	N/A
PB2	FLAG8	N/A

Table 1: Connections between host and devices

SOFTWARE DESIGN

All C code was written and compiled with CodeWarrior © V4.1 software that accompanies 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 HI3598_3599DEMO.c, attached and MC9S12XDT512.h, V2.02, which is available in the Freescale Demo Board. package.)

A short function was written for this application note software in order to simplify instructions. The routine enables 8-bit SPI data transfers between the Demo Board and the HI-3598 or HI-3599 while concurrently receiving an 8-bit byte from the Holt device. The following function name and argument list is used throughout the software program.

txrx8bits(char txbyte, char return): transfers 8 bit read/write data

A more detailed explanation of this function may be found in the attached example program, HI3598_3599DEMO.c.

An expanded header file, 3598DEMO.h, written to accompany the C program, is also attached. The header contains definitions for status register bits, control register bits and LEDs and switches.

The source code for both the C file (HI3598_3599DEMO.c) and header file (3598DEMO.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 0xN5hex, where 'N' specifies the channel number in hex.

The following code loads the control word:

```
// clear SPI status register
dummy = SPI0SR;
// writing opcode to Data Reg starts SPI xfer
SPI0DR = (j << 4) + 0x04;
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;
```

Loading Test Register

The self test register is loaded with one 32-bit word with op code 0xX8hex for fast speed and 0xX9hex for slow speed. The least significant bit of ARINC data must follow the last op code bit. For more information on this test register please see HI-3598, HI-3599 data sheet. If using an HI-3598 the ARINC word transmission is visible on pins TX1 and TX0.

The following code loads one word in the self test register:

```
// send op code (ignore returned data byte)
dummy = txrx8bits(0x08,1);  //or 0x09 for slow
// sending MS byte of variable data
dummy = txrx8bits((char)(TxBusWord>>24),1);
// send next byte (ignore returned data byte)
dummy =txrx8bits((char)((TxBusWord>>16)&0xFF),1);
// send next byte (ignore returned data byte)
dummy =txrx8bits((char)((TxBusWord>>8)& 0xFF),1);
// send LS byte (ignore returned data byte)
dummy = txrx8bits((char)(TxBusWord & 0xFF),1);
```

Unload Rx FIFO

If control word bit 1 = 0, FLAG will be high when the RX FIFO contains at least one valid ARINC word. Op code n3hex will retrieve one word at a time - where 'n' is the channel number.

The following code uses op code 0xN3hex, where 'N' specifies the channel number in hex, 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((k << 4) + 0x03,1);

// 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-3598 and HI-3599 have user-programmable label recognition for up to 16 different labels per channel. Op code n1hex, where 'n' is the channel number, is used to write 16 labels to a specific channel. Op code 0xN2hex, where 'N' is the channel number, is used to read the 16 programmed 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.

Writing label memory

Copy pre-assigned ARINC label selections from LabelArrayCh[ ][ ] to HI-3598 or HI-3599

```
// send op code (ignore returned data byte)
dummy = txrx8bits((j<< 4) + 0x01,1);

// send 16 bytes of ARINC label data
for (i=15; i>=0; I--) {
  // send 1 byte of label data,
  dummy = txrx8bits(LabelArrayCh[j-1][i],1);
}
```

Reading label memory

Verify match: HI-3598 or HI-3599 ARINC label selections to LabelArrayCh[ ][ ]

```
// send op code (ignore returned data byte)
rxbyte = txrx8bits((k << 4) + 0x02,1);

// starting at high end, read 8-bit increments of
// ARINC label data
for (i=15; i>=0; i--) {
 // send dummy data, read 1 byte of label data
 rxbyte = txrx8bits(0,1);
 // check for mismatch
 if (rxbyte != LabelArrayCh[k-1][i]) j=0x0000;}
```

STATUS REGISTER

The HI-3598 and HI-3599 have a 16-bit status register which can be polled to determine the status of the receiver FIFOs. Op code 0xX6hex is used to retrieve the status register bits.

The following code is an example of how to use op code x6 hex and store the contents in a variable to be used throughout the program.

```
// clear SPI status register
rxword = SPI0SR;
// writing opcode to Data Reg starts SPI xfer
SPI0DR = 0x06;
// wait for SPIF flag assertion
while (!SPI0SR_SPIF);
// read/ignore Rx data in Data Reg, resets SPIF
rxword = SPI0DR;
// send dummy data, receive upper Control Reg
SPI0DR = 0;
// wait for SPIF flag assertion
while (!SPI0SR_SPIF) { }
// read upper Control Reg byte in Data Reg
rxword = (SPI0DR<<8);
// send dummy data, receive lower Control Reg
SPI0DR = 0;
// wait for SPIF flag assertion
while (!SPI0SR_SPIF);
// read lower Control Reg byte in Data Reg
rxword = rxword|SPI0DR;
```

Additional Information

Information on the Freescale demonstration board, Code Warrior © and microcontroller can be found at www.Freescale.com.

EXAMPLE PROGRAMS

```
/*******************************************************************************
*
*       Copyright (C) 2008 Holt Integrated Circuits, Inc.
*       All Rights Reserved
*
* Filename:     3598DEMO.h
* Revision:     1.0
*
* Description: DEMO9S12XDT512 Board Header File
*
* Notes:       Used in Project 3598_3599.mcp
*              Created using CodeWarrior v4.1 for S12X.
*
*******************************************************************************/

/* include peripheral declarations */
#include <mc9s12xdt512.h>

/* 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 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 FLAG1 PORTA_PA0
#define FLAG PORTA_PA1
#define FLAG2 PORTA_PA4
#define FLAG3 PORTA_PA5
```

```
#define FLAG4 PORTA_PA6
#define FLAG5 PORTA_PA7
#define FLAG6 PORTB_PB0
#define FLAG7 PORTB_PB1
#define FLAG8 PORTB_PB2

#define MR    PORTA_PA2
#define MARK  PORTA_PA3



/* define Control Register bits */

// ACTION IF BIT IS SET    //  ACTION IF BIT IS *NEGATED*
//------------------------ // --------------------------------
#define RXSPEED_LOW 0x0001 // ARINC receive bus speed = high
#define RXFULL_FLAG 0x0002 // RFLAG output = Rx FIFO Empty
#define LABELS_ON   0x0004 // ARINC word label decoding is off
#define RX_RESET    0x0008 // RX reset, disabled if held high
#define RXPARITY_ON 0x0010 // receive parity off (all 32 bits = data)
#define LOOPBAK_OFF 0x0020 // Tx-to-Rx loop-back is enabled
#define RXDECODE_ON 0x0040 // decoding of Rx ARINC bits 10:9 is off
#define LBL_NOREV   0x0200 // Tx/Rx label bit order is reversed
// 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) 2008 Holt Integrated Circuits, Inc.
*       All Rights Reserved
*
* Filename:    HI3598_3599DEMO.c
* Revision:    1.0
*
* Description: Test Routines for Holt HI-3589 & HI-3599
*              ARINC 429 ICs with SPI Interface
*
* Notes:       Uses the Freescale DEMO9S12XDT512 board revision C.
*              Created using CodeWarrior v4.1 for S12X.
*
******************************************************************************/

#include <hidef.h>           // common defines and macros
#include <mc9s12xdt512.h>    // derivative information
#include <string.h>
#include "xgate.h"
#include "3598DEMO.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[8][4];

/* Initialize HI-3589/HI-3599 Control Register:  */
unsigned short ControlReg =LABELS_ON ;

/* Control Register bits to use:
   RXSPEED_LOW|RXFULL_FLAG|LABELS_ON
   RX_RESET|RXPARITY_ON|LOOPBAK_OFF
   LBL_NOREV

   RXBIT10_HI|RXBIT9_HI
*/

//====================================================================
/* Initial selections for active ARINC word labels
   LabelArrayCh [n][k]
   where [n] = Receiver number
         [k] = label number

   Example:
   LabelArrayCh [2][5] denotes receiver 2 label position 5 */

unsigned char LabelArrayCh [8][16] = {

{
// ------------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ------------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ------------------------------------------------------------------
},
{
// LabelArrayCh2[16] =
// ------------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ------------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ------------------------------------------------------------------
},
{
```

```
//LabelArrayCh3[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
},
{
// LabelArrayCh4[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
},
{
// LabelArrayCh5[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
},
{
// LabelArrayCh6[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
},
{
// LabelArrayCh7[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
},
{
// LabelArrayCh8[16] =
// ---------------------------------------------------------------
//   [0]     [1]     [2]     [3]     [4]     [5]     [6]     [7]
     0X01,   0X02,   0X03,   0X04,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
//   [8]     [9]     [10]    [11]    [12]    [13]    [14]    [15]
     0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,   0X00,
// ---------------------------------------------------------------
}
};

// ---------------------------------------------------------------------------
// 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..2] input (FLAG6-8)
    DDRB_DDRB1 = 0;
    DDRB_DDRB2 = 0;
    DDRB_DDRB3 = 0;           // Port B[3] input (SW3 4)
    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 - 4..7] input (FLAG & FLAG1-5))
    DDRA_DDRA1 = 0;
    DDRA_DDRA4 = 0;
    DDRA_DDRA5 = 0;
    DDRA_DDRA6 = 0;
    DDRA_DDRA7 = 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 = 0x20;//02
    // 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() */

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

Argument(s):  return_when_done

     Return:  nothing
     Action:  This function sends the 8 bit op code 0x07hex to reset
              the device using SPI0.

              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:  masterReset(1); // apply MR, return immediately */

void masterReset (unsigned char return_when_done) {

  unsigned char dummy;

  dummy = SPI0SR;          // clear SPI status register

  SPI0DR = 0x07;         // 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

}   /* masterReset */

/* ------------------------------------------------------------------
/  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-3589/HI-3599 Control Register Function
/  ------------------------------------------------------------------

Argument(s):  j = receiver number

     Return:  nothing
     Action:  Using SPI0, this function transmits op code 0xn4 where
              n is the receiver number 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 (unsigned short j) {

  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;
  //============================================

  dummy = SPI0SR;       // clear SPI status register
  SPI0DR = (j << 4) + 0x04;        // 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
  //============================================
  // de-assert 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-3589/HI-3599 Control Register Function
/  ---------------------------------------------------------------------

Argument(s):  j = receiver number

     Return:  16-bit Control Register value

     Action:  Using SPI0, this function transmits op code 0xn5 where
              n is the receiver number 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 (unsigned short j) {

  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 = (j << 4) + 0x05;          // 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
  //---------------------------------------------
  // de-assert 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-3589/HI-3599 Status Register Function
/  ------------------------------------------------------------------

Argument(s):  none

     Return:  8-bit Status Register data

     Action:  Using SPI0, this function transmits op code 0x06 plus
              1 byte of dummy data using two 8-bit SPI transfers.
              The received byte from SPI transfer #2 is returned  */

unsigned short readStatusReg (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 = 0x06;          // 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
  //---------------------------------------------


  // de-assert 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;

}/* readStatusReg() */
```

```
/* -----------------------------------------------------------------
/  Read HI-3589/HI-3599 next RxFIFO Word Function
/  -----------------------------------------------------------------

Argument(s):  k = receiver number

     Return:  next 32-bit ARINC word in RxFIFO

     Action:  Using SPI0, this function transmits op code 0xn3 where
              n is the receiver number 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

              If RxFIFO is empty, returns 0x00000000 */

unsigned long read1RxFIFO (unsigned short k) {

  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 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((k << 4) + 0x03,1);
  // read up to 32 ARINC 32-bit data words

  // 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 ;

  // de-assert 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;

}/* read1RxFIFO() */


/* -----------------------------------------------------------------
/  Write HI-3589/HI-3599 TxFIFO Function
/  -----------------------------------------------------------------

Argument(s):  TxBusWord = the word to be transmitted

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x08. 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:  writeTxSlow(0xABCD1234); // writes a word to Tx register from the
                                 //  sending which is transmitted on TX1 and TX0
                                 //  as well as internally in self test mode */

void writeTxSlow (unsigned long TxBusWord) {
  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;
  // send op code (ignore returned data byte)
  dummy = txrx8bits(0x09,1);

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

  // de-assert 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;
}/* writeTxSlow() */


/* ------------------------------------------------------------------
/  Write HI-3589/HI-3599 TxFIFO Function
/  ------------------------------------------------------------------

Argument(s):  TxBusWord = the word to be transmitted

     Return:  none

     Action:  Using SPI0, this function transmits op code 0x08. 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:  writeTxFast(0xABCD1234); // writes a word to Tx register from the
                                 //  sending which is transmitted on TX1 and TX0
                                 //  as well as internally in self test mode */

void writeTxFast (unsigned long TxBusWord) {

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

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

  // de-assert 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;
}/* writeTxFast() */
```

```
/* ----------------------------------------------------------------
/  Copy HI-3589/HI-3599 ARINC label selections from LabelArrayCh[]
   to HI-3589/HI-3599
/  ----------------------------------------------------------------

Argument(s):  j = channel number

     Return:  none

     Action:  Using SPI0, this function transmits op code 0xn1 where n
              is the receiver number. This function then copies the
              global LabelArray[n] to HI-3589/HI-3599 label memory
              so labels are enabled/disabled to match the
              program's LabelArrayCh[].                            */

void copyAllLabels (unsigned short j ) {

  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((j<< 4) + 0x01,1);
  // send 16 bytes of ARINC label data
  for (i=15; i>=0; i--) {
    // send 1 byte of label data, ignore returned data byte
    dummy = txrx8bits(LabelArrayCh[j-1][i],1);
  }

  // de-assert 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-3589/HI-3599 ARINC label selections to LabelArray[]
/  ----------------------------------------------------------------

Argument(s):  k = receiver number

     Return:  0xFFFF if LabelArrayCh[n][k] fully matches HI-3589/HI-3599 label memory

     Action:  Using SPI0, this function transmits op code 0xn2 where n
              is the receiver number. This function then compares all
              16 labels in HI-3589/HI-3599 label memory (8 bits at a
              time) to the corresponding element in the program's
              LabelArrayCh[n]. Return value indicates whether or not
              mismatch is detected.                                 */

unsigned short checkAllLabels (unsigned short k) {

  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((k << 4) + 0x02,1);
  j = 0xffff;
  // starting at high end, read 8-bit increments of ARINC label data
  for (i=15; i>=0; i--) {
    // send dummy data, read 1 byte of label data
    rxbyte = txrx8bits(0,1);
    if (rxbyte != LabelArrayCh[k-1][i]) j=0x0000;
  }

  // no errors, de-assert 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 "pass or fail" value
  return j;
} /* checkAllLabels */





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

  volatile unsigned short rxword;
  unsigned short rxwordlab;
  unsigned long i,i1=0,i2=0,i3=0,i4=0,i5=0,i6=0,i7=0,i8=0;

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

  // apply HI-3598/HI-3599 Master Reset.
  // by software, return only after completion...
  masterReset (1);
  // or, by hardware, HI-3598 only, set MR , delay, reset MR...
  //MR = 1;
  //for (i = 100; i > 0; i--) ;
  //MR = 0;


  ControlReg =LABELS_ON ;

  // initialize the HI-3598 Control Registers
  writeControlReg(0x01);
  writeControlReg(0x02);
  writeControlReg(0x03);
  writeControlReg(0x04);
  writeControlReg(0x05);
  writeControlReg(0x06);
  writeControlReg(0x07);
  writeControlReg(0x08);



  // read back HI-3598 Control Register
  rxword = readControlReg(0x01);
  // update LED1 to show match or mismatch
  if (rxword == ControlReg) LED1 = ON;
  else LED1 = OFF;

  rxword = readControlReg(0x02);
  rxword = readControlReg(0x03);
  rxword = readControlReg(0x04);
  rxword = readControlReg(0x05);
  rxword = readControlReg(0x06);
  rxword = readControlReg(0x07);
  rxword = readControlReg(0x08);

  copyAllLabels(0x01);
```

```
rxwordlab = checkAllLabels(0x01);
if (rxwordlab == 0xFFFF) LED2 = ON;
      else LED2 = OFF;

// update LED2 to show label read-back result,
// match (FFFF) or mismatch (any other value)

copyAllLabels(0x02);
rxwordlab = checkAllLabels(0x02);
copyAllLabels(0x03);
rxwordlab = checkAllLabels(0x03);
copyAllLabels(0x04);
rxwordlab = checkAllLabels(0x04);
copyAllLabels(0x05);
rxwordlab = checkAllLabels(0x05);
copyAllLabels(0x06);
rxwordlab = checkAllLabels(0x06);
copyAllLabels(0x07);
rxwordlab = checkAllLabels(0x07);
copyAllLabels(0x08);
rxwordlab = checkAllLabels(0x08);

// read back the HI-3598/HI-3599 Status Register
rxword = readStatusReg();

writeTxFast(0x01020304);
for (i = 0x2FF; i > 0; i--) ;
//wait for transmit

writeTxFast(0x02030405);
for (i = 0x2FF; i > 0; i--) ;

writeTxFast(0x03040506);
for (i = 0x2FF; i > 0; i--) ;

writeTxFast(0x04050607);
for (i = 0x2FF; i > 0; i--) ;

while (FLAG){

  while (!(readStatusReg()&0x0001)) {   // Check status register for receiver 1
     RxBusWord[0][i1]=read1RxFIFO(0x01); // if FIFO contains data, empty contents
     i1++;
     if (i1 ==4)
      i1=0;
  };

  while (!(readStatusReg()&0x0002)) {   // Check status register for receiver 2
     RxBusWord[1][i2]=read1RxFIFO(0x02); // if FIFO contains data, empty contents
     i2++;
     if (i2 ==4)
      i2=0;
  };

  while (!(readStatusReg()&0x0004)) {   // Check status register for receiver 3
     RxBusWord[2][i3]=read1RxFIFO(0x03); // if FIFO contains data, empty contents
     i3++;
     if (i3 ==4)
      i3=0;
  };

  while (!(readStatusReg()&0x0008)) {   // Check status register for receiver 4
     RxBusWord[3][i4]=read1RxFIFO(0x04); // if FIFO contains data, empty contents
     i4++;
     if (i4 ==4)
      i4=0;
  };

  while (!(readStatusReg()&0x0010)) {   // Check status register for receiver 5
     RxBusWord[4][i5]=read1RxFIFO(0x05); // if FIFO contains data, empty contents
     i5++;
     if (i5 ==4)
      i5=0;
  };
```

```
      while (!(readStatusReg()&0x0020)) {   // Check status register for receiver 6
         RxBusWord[5][i6]=read1RxFIFO(0x06); // if FIFO contains data, empty contents
         i6++;
         if (i6 ==4)
          i6=0;
      };

      while (!(readStatusReg()&0x0040)) {   // Check status register for receiver 7
         RxBusWord[6][i7]=read1RxFIFO(0x07); // if FIFO contains data, empty contents
         i7++;
         if (i7 ==4)
          i7=0;
      };

      while (!(readStatusReg()&0x0080)) {   // Check status register for receiver 8
         RxBusWord[7][i8]=read1RxFIFO(0x08); // if FIFO contains data, empty contents
         i8++;
         if (i8 ==4)
          i8=0;
      };
    }

    for (;;) {
    // toggle LED periodically to show activity
      i++;
      if (i & 0x20000) LED4 = 1;
      else LED4 = 0;
    }// Loop forever

  }//end main(void)
```

Figure 1: Example Circuit for HI-3598PQx

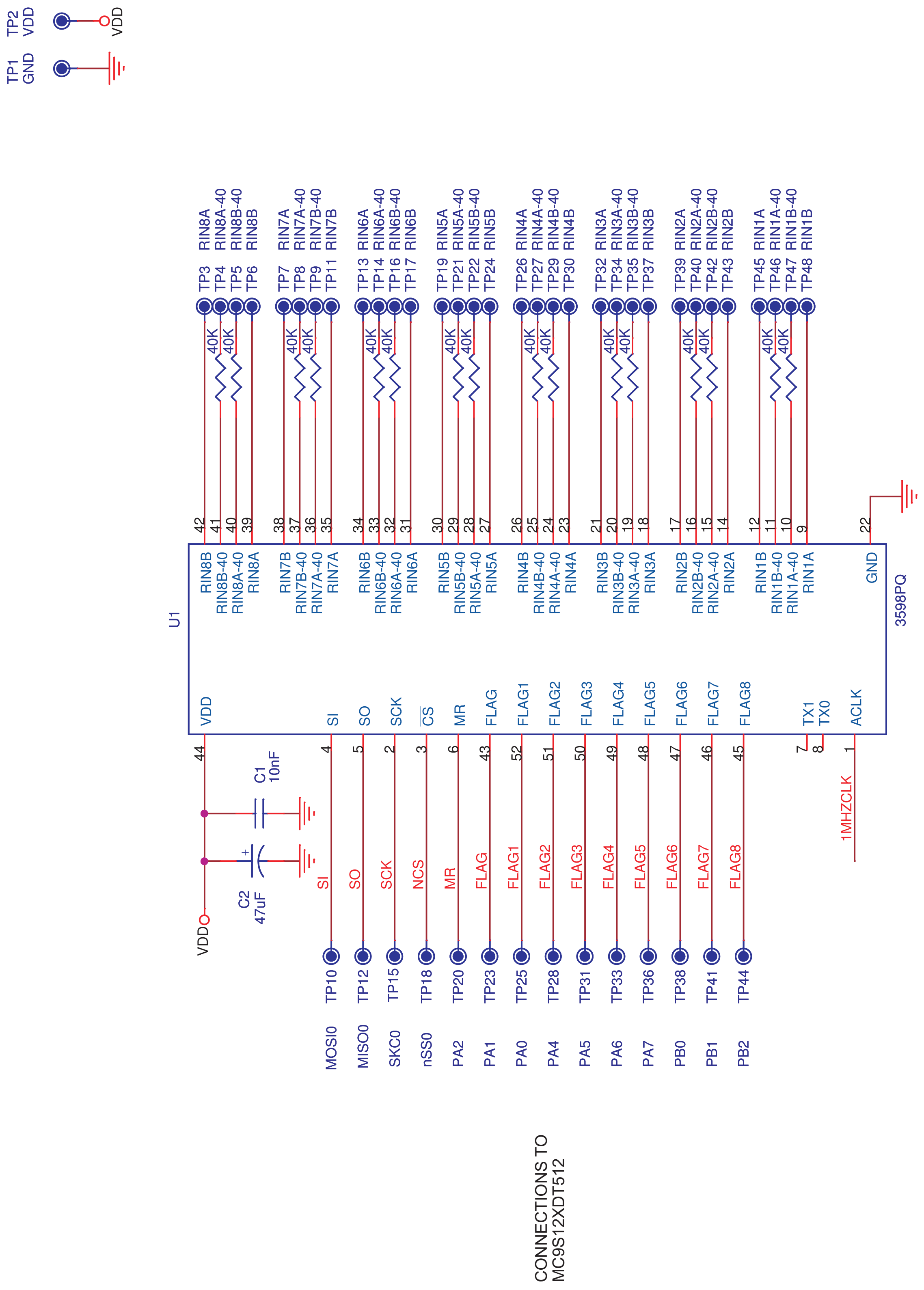
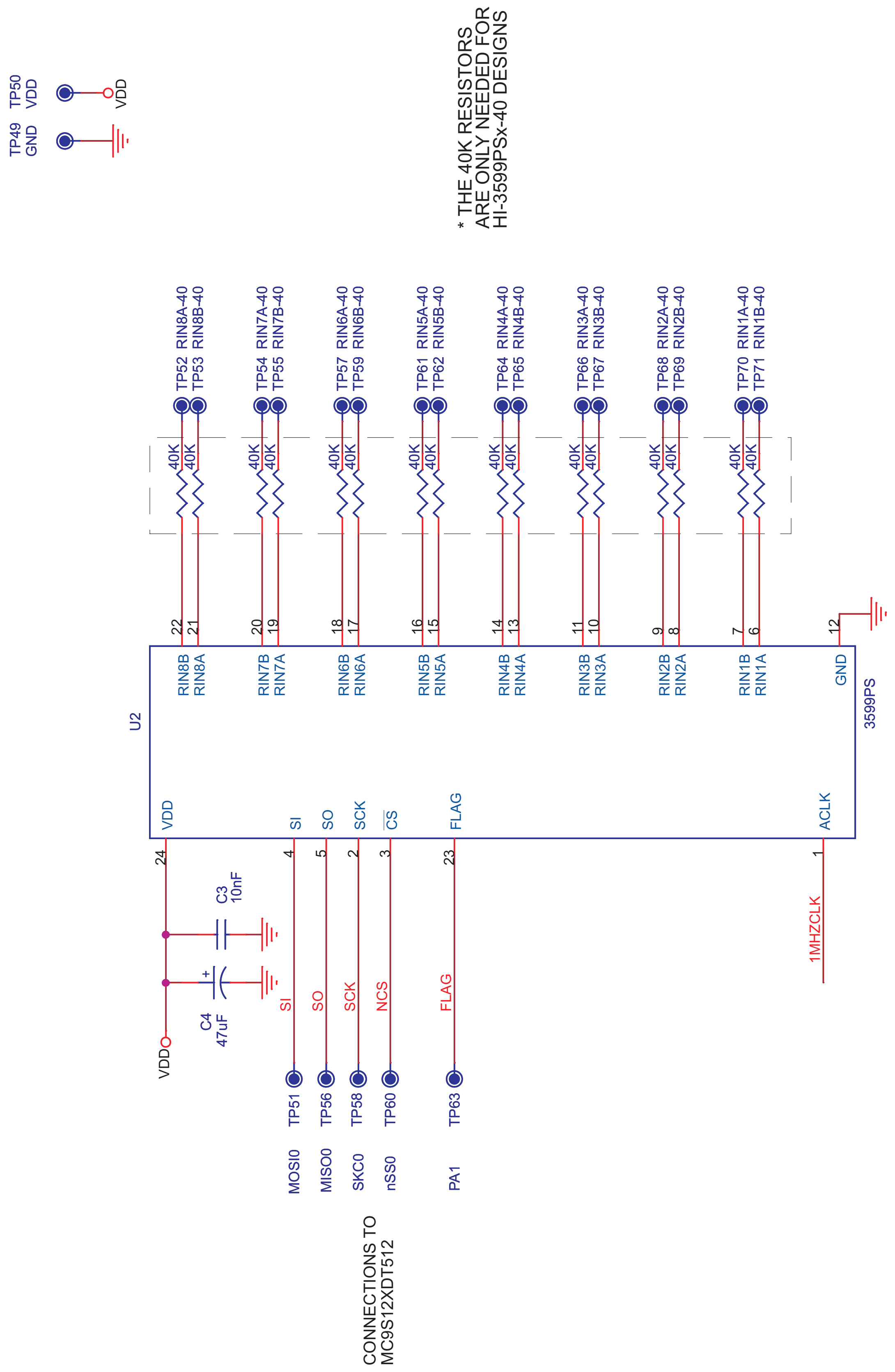


Figure 2: Example Circuit for HI-3599PSx or HI-3599PSx-40



REVISION HISTORY

Revision	Date	Page	Description of Change
AN-150, Rev. NEW	07/02/08	All	Initial Release