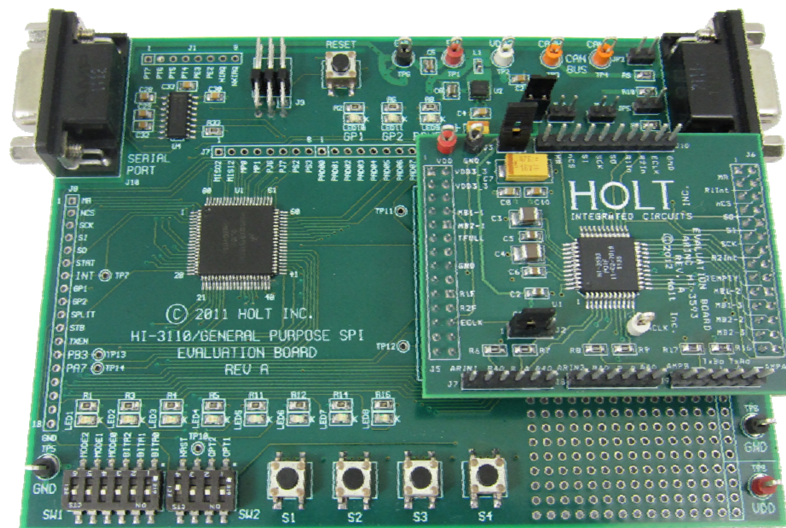
HOLT INC.
INTEGRATED CIRCUITS

23351 Madero,
Mission Viejo, CA
92691. USA.
Tel: + 1 949 859 8800
Fax: + 1 949 859 9643
Email: sales@holtic.com
Web: www.holtic.com

HI-3593 ARINC 429 3.3V Dual Receiver, Single Transmitter with SPI
Application Note AN-161

June 13, 2012



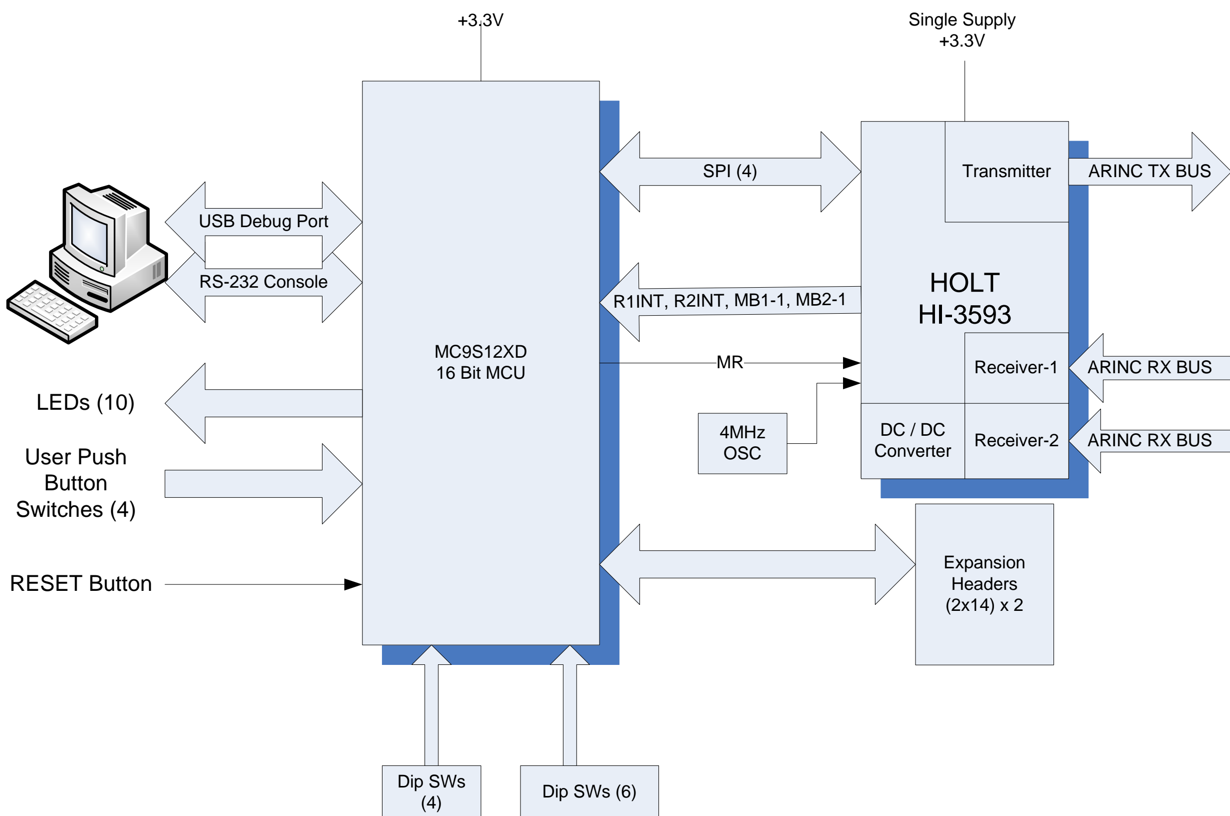
INTRODUCTION

This application note provides more detail on the HI-3593 demo software provided in the Holt HI-3593 ARINC 429 Evaluation Kit. The main sections of this application note are:

- Demo software overview
- Demo Project setup with Freescale CodeWarrior.

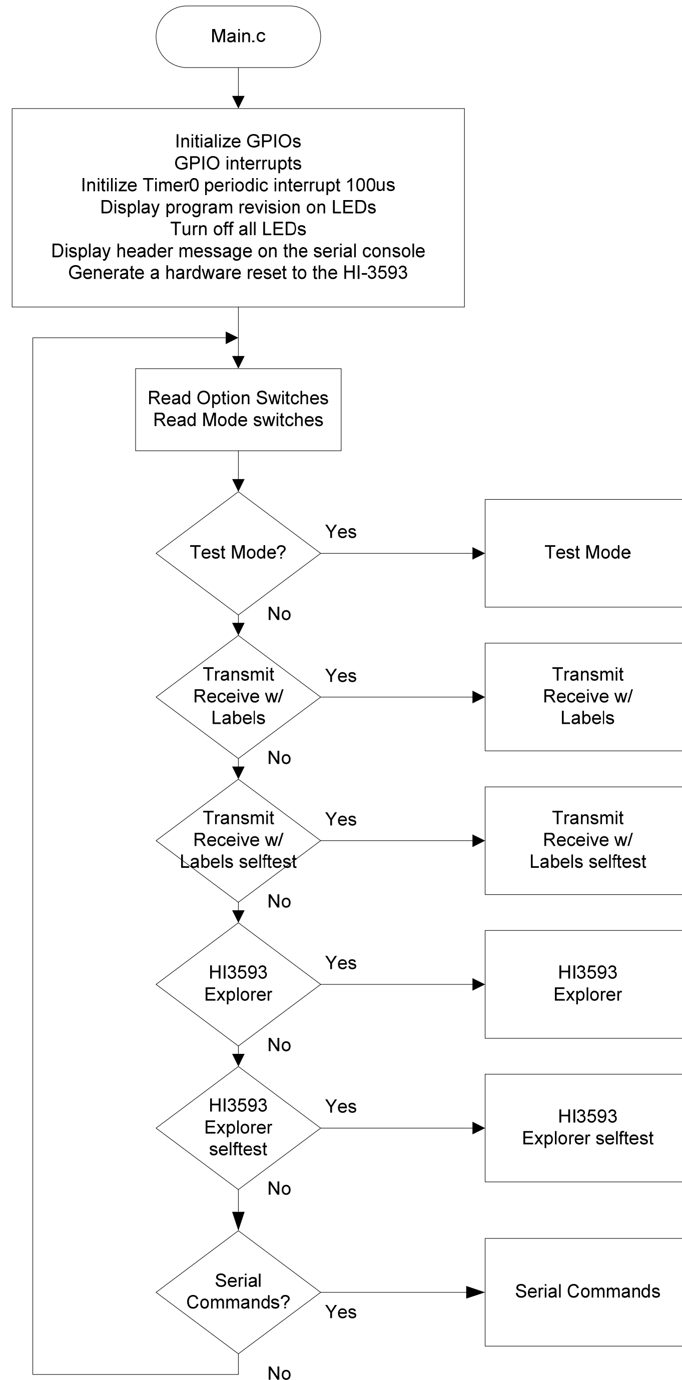
A Quick Start Guide and a User's Guide for HI-3593 evaluation kit can be found on the CD-ROM. Use these guides to become more familiar with the board setup and operation of the demo software.

Evaluation Board Block Diagram



Demo software overview

This overview flow chart shows the demo program at a glance.



The program enters the desired mode selected by the mode switches. To restart into a different mode, reconfigure the mode switches and reset the board.

MCU Clock and SPI Frequencies

The Freescale MC9S12XDT512 (MCU) on the main board uses a 4MHz crystal for operation and the built-in PLL multiplies this by 20 to achieve an 80MHz system clock. This system clock is divided by two for a 40MHz Bus Clock which is used internally for the MCU peripherals.

The PLL is programmed to multiply by 20 by this line of code in the Peripherals.c module:

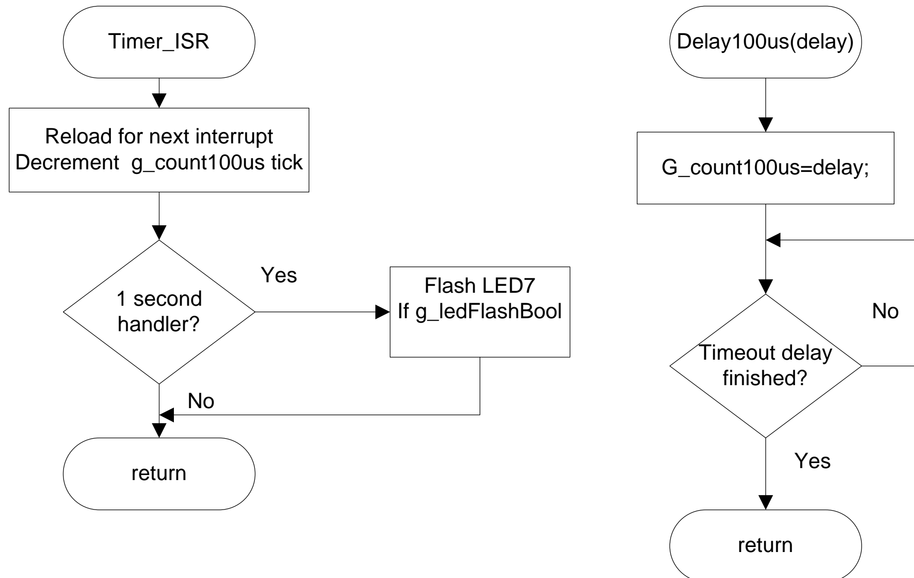
```
SYNR = 9;                  // 80Mhz PLL system clock
```

The SPI frequency is set by this line of code in the Peripherals.c module:

```
SPI0BR = SPI_5MHZ;         // SPI CLK = 5MHz (see "Peripherals.c" for other
                           // rates)
```

The maximum SPI frequency for the HI-3593 is 10MHz.

Timing and Delay Functions



These functions provide the basic timing for the program. The Delay100us() can be used anywhere an accurate delay is needed in the program .

The global g_count100us variable is decremented at the 100us timer rate. This variable is used by a general delay function which can be called with a specified number of delay intervals. The g_count100us variable is a 16-bit integer so the delay ranges from 100us to 6.5536 seconds.

```
// -------------------------------------
// General timer tick 100us for delays
// -------------------------------------
void Delay100us(unsigned int delay){
    g_count100us=delay;
    while(g_count100us);

}
```

A number of predefined constants are defined which can be used by calling the function with these constants.

```
#define K_1MS    10         // 1ms
#define K_10MS   100        // 10ms
#define K_100MS  1000       // 100ms
#define K_1SEC   10000      // 1 second
```

```
Usage: Delay100us(K_1SEC); // delay for one second
```

A one second interrupt handler in the TIMER_ISR is provided. Any code placed here automatically executes every second.

```
if(!count100us)
  {                                 // 1 second scheduler
      count100us = K_1SEC;
      if(ON==g_ledFlashBool)        // Flash the LED7 if enabled
        LED7 ^= TOGGLE;             // Alive 1 second blink
  }
```

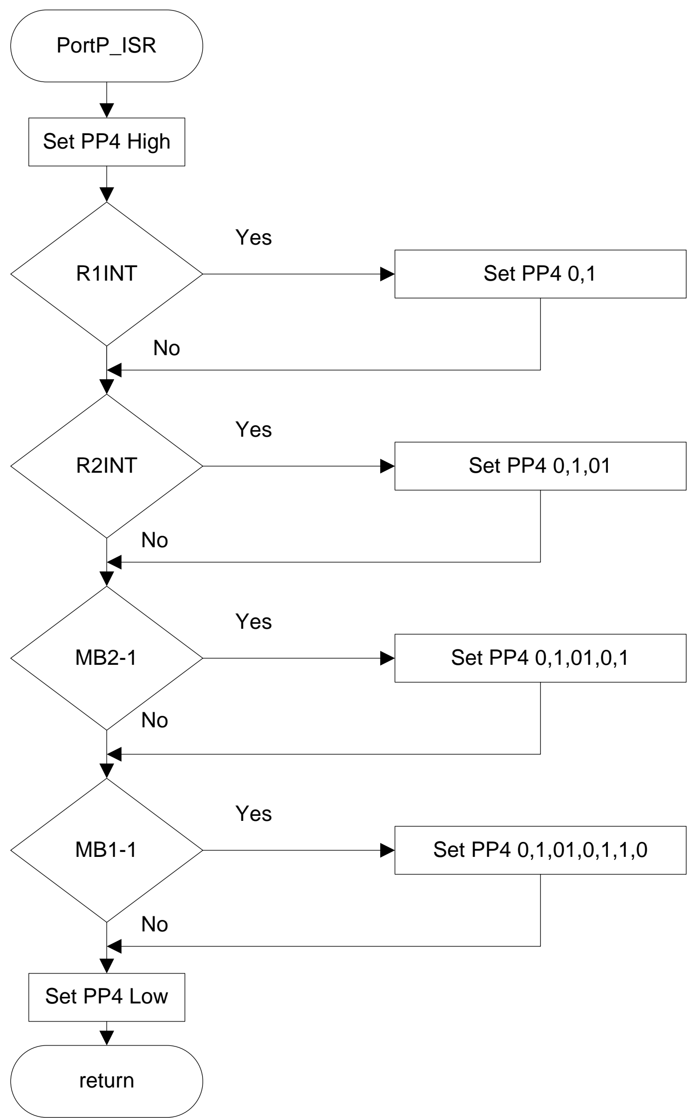
GPIO interrupt Handlers

An interrupt handler manages most of the interrupt pins on the 3593.

```
interrupt 56 void PORTP_ISR(void)
```

There are reserved interrupt handlers for 3593 interrupt output pins R1INT, R2INT, MB1-1 and MB2-1 already implemented in the function. User code can be implemented in these handlers to manage these interrupts.

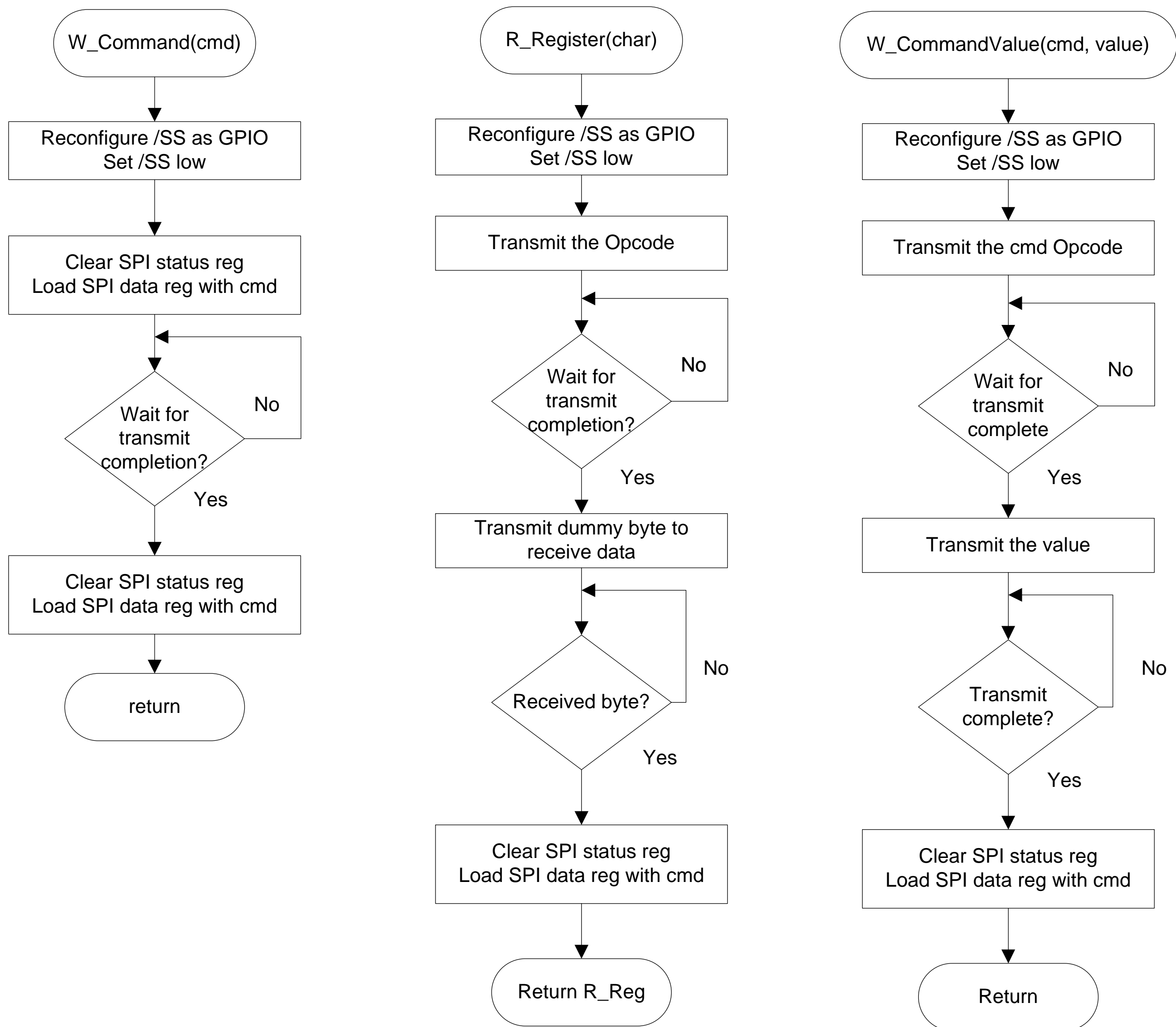
CAUTION: if code added to the interrupt handlers accesses the SPI interface, then any SPI access in the foreground code must be made anatomic by first disabling interrupts then re-enabling interrupts after the SPI is accessed. Failure to protect foreground SPI access will most likely cause corrupted data on the SPI interface. There are some test instructions in the handlers which pulse the PP4 MCU pin (MISO2) J7-1 on the main board a number of times as a debugging tool. By viewing these pulses on an oscilloscope, the interrupt handler can be identified. This test code can be removed.



Pulse PP4 the number of times specified to identify the interrupt handler in process. This is just a debugging tool and can be removed by the user.

SPI Driver Functions

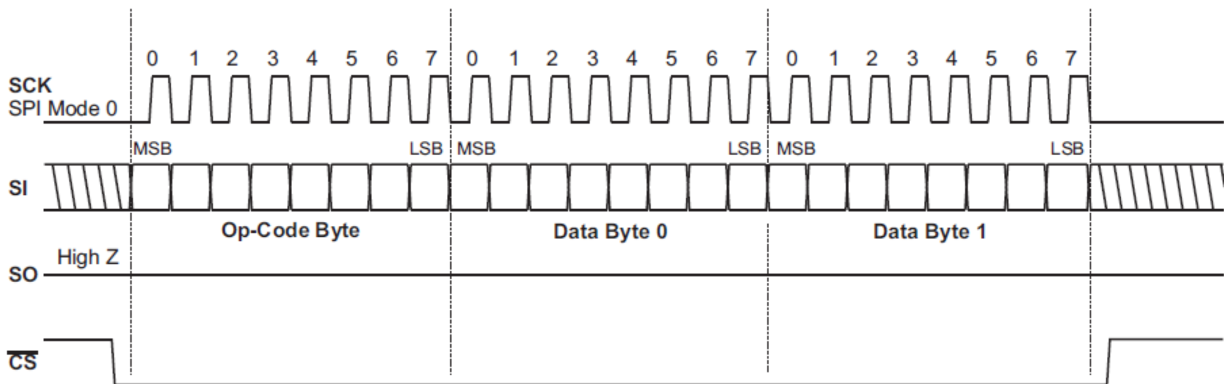
These three primitive SPI functions make up the basic read and write functions to access the SPI interface of the 3593. There are slightly more complicated functions to perform multi-byte reads or writes which are basically derivatives of these three simpler functions. All 3593 SPI driver functions are included in the 3593Driver.c module and its 3593Driver.h header file. The MCU /SS pin is connected to the 3593 /CS pin.



Special handling of the /SS SPI signal:

All 3593 SPI Op-Codes require the /CS to remain low for the complete duration of the data transfer including multi-byte reads and writes. Refer to figures 6 and 7 of the data sheet for timing diagram examples.

To achieve this, the default SPI slave select line /SS in the Freescale MCU must be reconfigured as a GPIO and controlled by code in the function. This technique is common for devices requiring the /CS line to remain low during multi-byte transfers. The first positive SCK edge must occur after /CS is asserted low; the last falling SCK edge must occur before the /CS is negated high as shown in the following diagram:



There are functions that read a single byte from the 3593 SPI port, write a command to the 3593 SPI port and a few others which read or write a command plus a multiple number of bytes. For example the function below is the basic function to write out a command plus one byte of data to the 3593 SPI port.

```
// Write SPI Command with a Value to HI-3593
void W_CommandValue (uint8 cmd, uint8 value){
  uint8 dummy;

  SPI0CR1 = SPI0CR1 & ~SPI0CR1_SSOE_MASK;   // disable auto /SS output, reset /SS Output
  SPI0CR2 = SPI0CR2 & ~SPI0CR2_MODFEN_MASK; // disable auto /SS output, reset SPI0 Mode
  SPI0_nSS = 0;                             // assert the SPI0 /SS strobe
  dummy = SPI0SR;                           // clear SPI status register
  SPI0DR = cmd;                             // SPI command
  while (!SPI0SR_SPIF);
  dummy = SPI0DR;                           // read Rx data in Data Reg to reset SPIF
  dummy = SPI0SR;                           // clear SPI status register
  SPI0DR = value;                           // Reset values
  while (!SPI0SR_SPIF);
  dummy = SPI0DR;                           // read Rx data in Data Reg to reset SPIF

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

This function is used to transmit a command byte followed by four data bytes to write one ARINC message to the FIFO.

```
// ---------------------------------------------------------------
// Transmits the Message Command and data contained in the passed array pointer
// Transmit the 0x0C Opcode command + 4 bytes of
// ARINC data per Figure 1, pg 8 of the data sheet
// ---------------------------------------------------------------
 void TransmitCommandAndData(uint8 cmd, uint8 *TXBuffer)
 {
 uint8 static ByteCount,dummy,transmitCount;

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

   transmitCount=4;             // Standard messages are 4 bytes if writing PL Match registers
                                //  send only 3 bytes
   if(cmd==W_PL1Match || cmd==W_PL2Match)
      transmitCount--;

   dummy = txrx8bits(cmd, 1);   // Transmit the whole message,ignore return values
   // Transmit command=0x0C +  4 bytes
   for(ByteCount=0; ByteCount< transmitCount; ByteCount++)
      {
         // Transmit the whole message, ignore return values
         dummy = txrx8bits(TXBuffer[ByteCount], 1);
      }
   SPI0_nSS = 1;                  // negate the SPI0 /SS strobe
   SPI0CR1 |= SPI0CR1_SPE_MASK|SPI0CR1_SSOE_MASK;
   SPI0CR2 |= SPI0CR2_MODFEN_MASK;

}
```

This example shows a very simple SPI driver function which issues one Opcode then Reads back one byte. This is used for fetching single byte status' from the SPI.

```
/* -----------------------------------------------------------------
/  Read HI-3110 Register Read Function
/  -----------------------------------------------------------------

Argument(s):  Register to read

     Return:  8-bit Register Value
*/

  unsigned char R_Register (char Reg) {
  unsigned char R_Reg;

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

The Init3593() function initializes the 3593 by first issuing a SPI master reset then initializing the ACLK divisor to divide the clock input by four. The ACLK oscillator module on the daughter card is 4MHz so this generates a 1MHz clock required by the 3593 needed to meet ARINC timings. Finally, the transmit status register (TSR) is fetched to check for 0x01 which is the expected value after a reset. If the calling function does not receive this expected value, the program turns on the red LED, transmits an error message on the console and enters a dead loop.

```
// ------------------------------------------------------------------
// Initialize the HI-3593
// ------------------------------------------------------------------
uint8 Init3593(uint8 AclkDiv, uint8 tmode, uint8 selftest, uint8 arate, uint8 tflip )
   {
   unsigned char cmd=0;

   W_Command(RESETCMD);                  // Reset the HI-3593
   W_CommandValue(DivReg, AclkDiv);      // ACLK div/4 divisor
   cmd =  arate;
   cmd |= selftest << 4;
   cmd |= tmode << 5;
   cmd |= tflip << 6;                   // TFLIP on
   W_CommandValue(TCR, cmd);            // Program the Transmit Control Register
   return R_Register (R_TSR);

   }
```

See the 3593Driver.h header file for the options available in the define statements.

Uart.c   Serial Port (RS-232)

The drivers to support the serial port (Console) are contained in this module. There are some function drivers to allow messages to be sent and received on the UART. This is useful to log status or data messages on HyperTerminal or any other terminal program. It currently uses polling to determine when the data receive or transmit registers can be read or written.

GPI and GP2

These two pins on the main board are renamed as MB1-1 and MB2-1 respectively. They will be low most of the time for this demo, so both of these LEDs, LED10 and LED11 will be on most of the time.

LEDs   LED1-LED8

These LEDs are controlled by a function in the program. LED1-LED4s and LED8 are low true logic and LED5-LED7s are high true logic. Using this support function allows a universal way to turn the LEDs on and off from the program. The Freescale MC9S12DT part uses the pins PE5, PE6, PE7 for configuration sense pins during reset, so the logic on these three pins need to be reversed so the MCU sees a low at reset time.

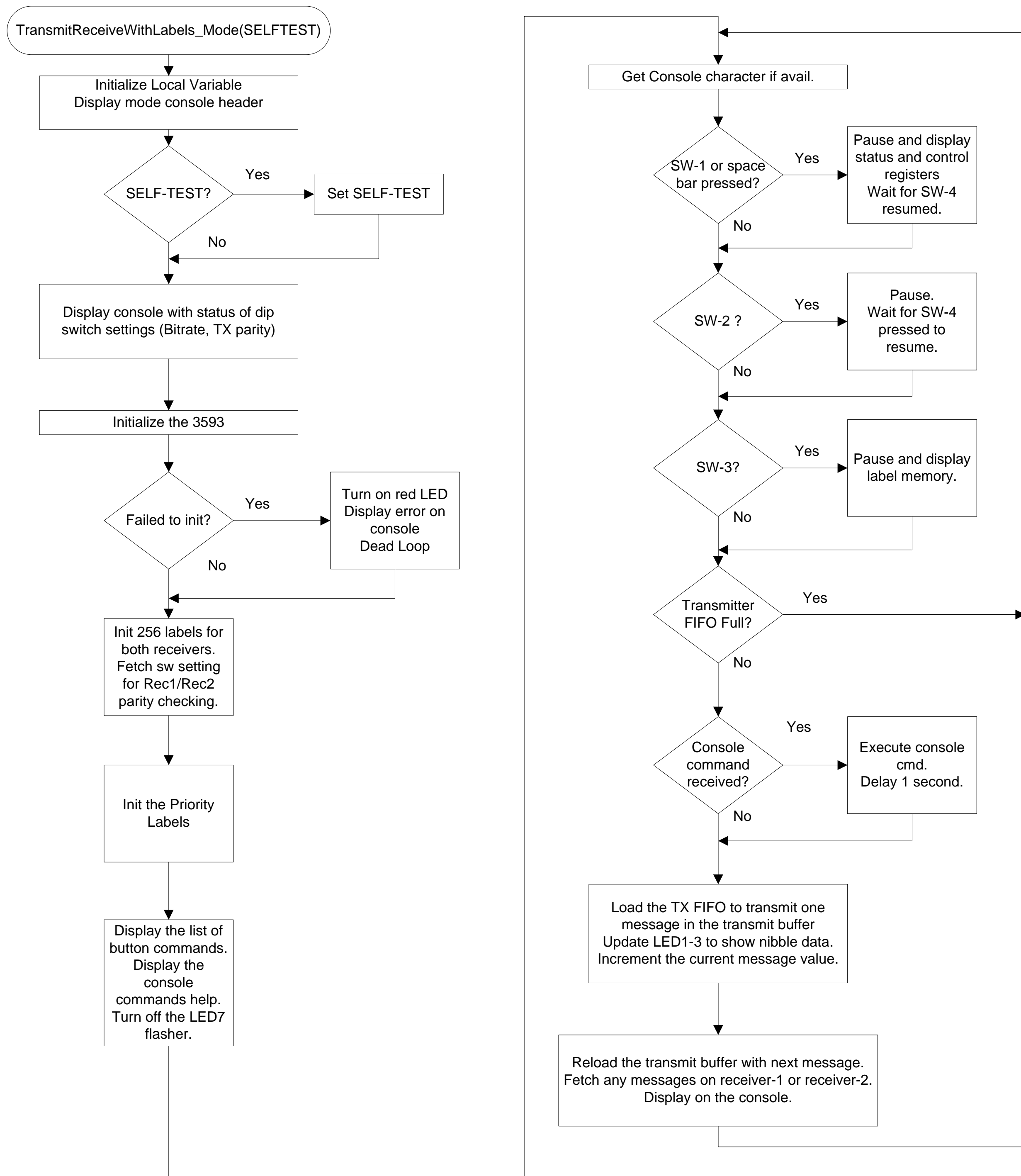
```
// -------------------------------------------------------
// Control LED1 - LED8
// ledNumber: LED_1,LED_2,LED_3,LED_4....LED_8 [1-8]
// OnOff: 1=ON, 0=OFF
// -------------------------------------------------------
void LED_CTL(uint8 ledNumber, uint8 OnOff){
#if NEWBOARD
   if(ledNumber>4 && ledNumber<8)// LEDs 5-7 have reversed HW logic so invert
                                        these 3
#else
   if(ledNumber>4)          // Old board.
#endif

      OnOff = ~OnOff;
   switch (ledNumber){
      case 1:  LED1=OnOff;  break;
      case 2:  LED2=OnOff;  break;
      case 3:  LED3=OnOff;  break;
      case 4:  LED4=OnOff;  break;
      case 5:  LED5=OnOff;  break;
      case 6:  LED6=OnOff;  break;
      case 7:  LED7=OnOff;  break;
      case 8:  LED8=OnOff;  break;
      default: break;
   }
}
```

Usage examples:

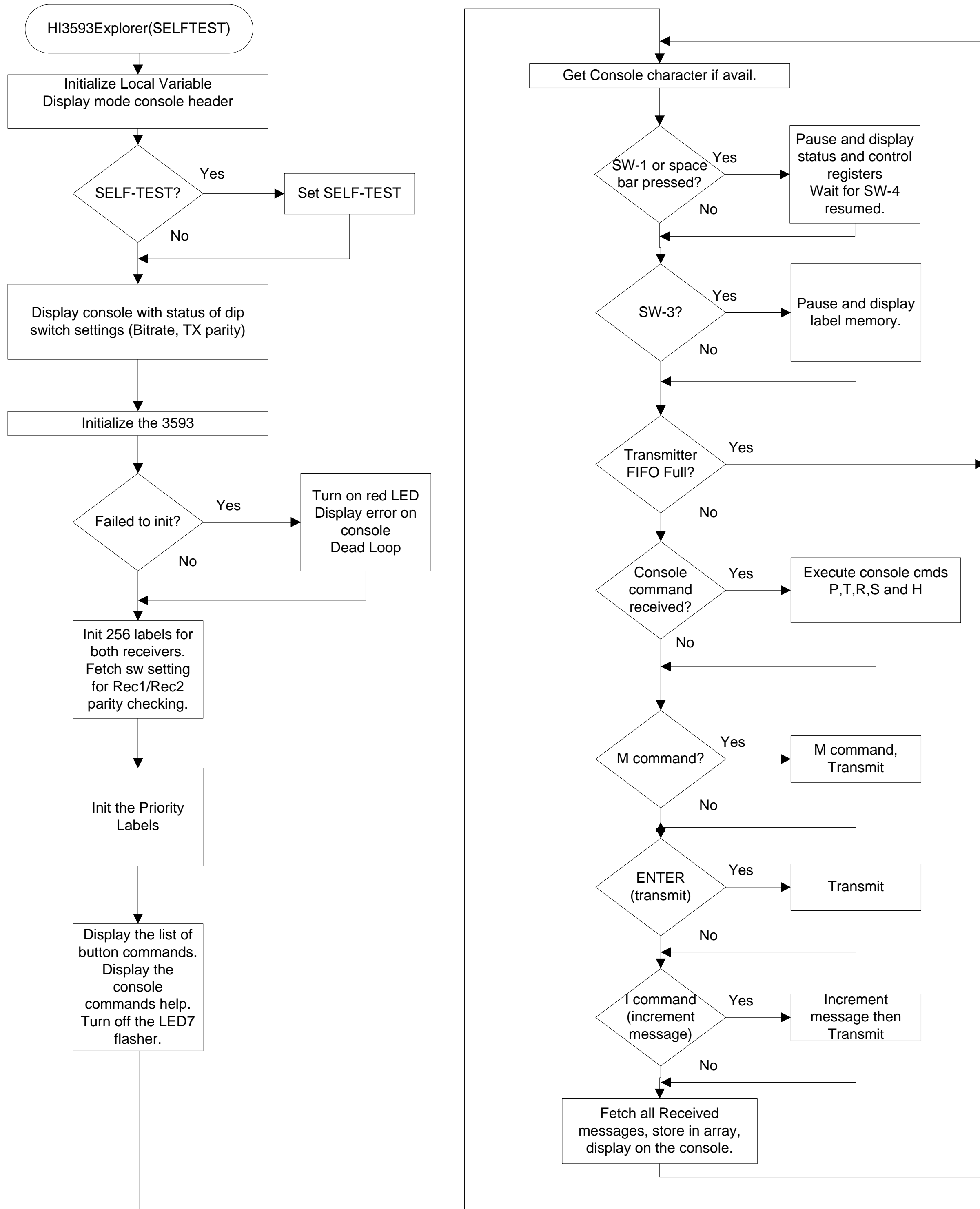
```
LED_CTL(LED_1,OFF);        // turns off LED1
LED_CTL(LED_1,ON);         // turns on LED1
```

The following two flow-charts illustrate the program flow for the two main demo modes.

Flow-chart of Mode-1 (Transmit Messages with Labels)



Flow-chart of Mode- 3 (HI3593Explorer)



HI-3593 demo Codewarrior Software Project

The software project is built with Freescale's CodeWarrior version 5.9.0 using the free limited 32K version. The current code size of the demo is approximately 16K. The main functions are in main.c and the low level HI-3593 drivers are in the 3593Driver.c file. The software project "HI-3593 Demo" will normally be distributed in a zip file on a CD-ROM with the same name. **To develop, debug and download this software into the board a PE Micro "USB Multilink Interface" debug cable is necessary. It is not provided in this kit.** To purchase this cable, go to the PE Micro website or purchase it from Digi-Key. See the links at the end of this document.

Project Files

Source Files

File	Description
main.c	Main code
3593Driver.C	SPI low-level drivers for the HI-3593
Peripherals.c	GPIO, PLL frequency setup and SPI configuration
BoardTest.c	Board Test functions
Uart.c	Low-level UART drivers
datapage.c	Freescale IDE support file

Include Files

File	Description
Main.h	
3593Driver.h	HI-3593 header
Peripherals.h	
BoardTest.H	
Uart.h	
Common.h	Common defines for the project
Derivative.h	Freescale IDE support file
Mc9s12xdt512.h	Freescale IDE target part support file

CodeWarrior and Software Project Setup:

1. Download and install the CodeWarrior IDE from the Freescale website. The download links are provided below.

2. Unzip the HI-3593 zip file into the directory you plan to use for your project.

3. Navigate to the HI-3593 project folder and double click the HI-3593 Demo.mcp project file to launch this project with CodeWarrior. The IDE should open with the project files on the left side of the window.

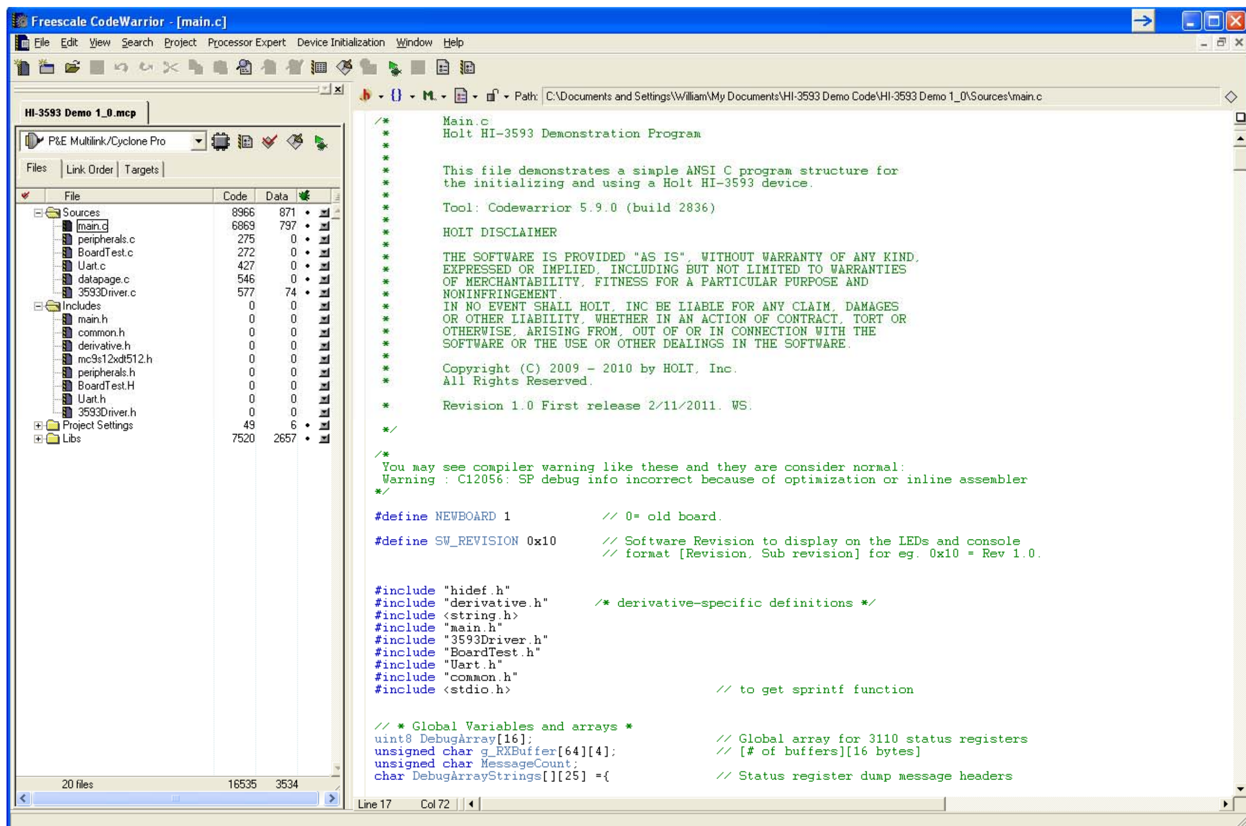
4. Click Make from the Project menu to rebuild the project. The project should build without errors. You may receive a dead assignment warning if for example some defines are set to a zero value.

5. Install the PE Micro USB Multilink Interface cable per the instructions.

6. Plug the USB Multilink 6-pin debug cable into the J9 debug connector and power up the board with 3.3V.

7. Download the program by clicking Debug from the Project menu. The first time you download you may need to configure the debugger for the USB Multilink cable. After downloading is complete the debugger window should be displayed with the first line in main.c highlighted. Press the green arrow button to run the program. Since the program has been loaded you can power down the board and re power the board and the program should run automatically without the debugger.

Holt HI-3593 project loaded with CodeWarrior 5.9.0



Freescale MC9S12XDT512xxx Development Tools

The Freescale microcontroller data sheet and other documentation can be found at this link:

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=S12XD&tid=16bhp

If these links become out of date go to: http://www.freescale.com/

and search for information on “S12XD: 16-Bit Automotive Microcontroller”.

A Free 32K limited version of the Code Warrior IDE from Freescale is available:

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CW-HCS12X&fsrch=1

The US Multilink debugger cable used for this project is:

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=USBMULTILINKBDM&parentCode=S12XD&fpsp=1

http://search.digikey.com/scripts/DkSearch/dksus.dll?Detail&name=USBMULTILINKBDME-ND

References:

http://www.holtic.com/

REVISION HISTORY

Revision	Date	Description of Change
AN-161, Rev. New	2-11-10	Initial Release
AN-161, Rev. A	6-13-12	Update board photo