Si57x/598/599 ANSI C Reference Design with Optional Non-Volatile Output Frequency

1. Introduction

Because each Si570/Si571/Si598/Si599 programmable XO/VCXO has a unique crystal frequency, it is necessary to perform the frequency conversion calculations for every new desired output frequency for each programmable oscillator. To simplify the creation of the search procedures and calculations, Silicon Laboratories is providing the following ANSI C-based reference design. These calculations can be performed with a variety of hardware and software; so, the ANSI C programming language is used. Because ANSI C is supported on a large number of processors, the code can be ported to nearly any platform.

The equations needed to change the output frequency involve multiplication and division and require sufficient precision to maintain the frequency accuracy. Two approaches are shown within the code. One approach uses double floating point precision and achieves better than 5 ppm accuracy. The alternate approach uses single floating point precision and yields better than 100 ppb. This method can be used with processors that do not support double floating point precision calculations.

Each customer will need to decide whether to make the calculations in real time during normal operation or to implement them during an outgoing test procedure with the result stored on an EEPROM or similar storage device.

If an independent processor is not already available within the design, the real-time calculations can be made by the simple addition of an external MCU such as the C8051F301 from Silicon Labs. The C8051F301 is 3x3 mm and contains sufficient Flash memory for this implementation. A strobe input on the C8051F301 is used to cycle through the list of stored frequencies (see Figure 1 on page 3 for the reference design expected usage model). The code examples in Appendix A and Appendix B both cycle through the list of frequencies, but the code in Appendix B stores the last output frequency in Flash memory and will start from the last output frequency after the power is cycled. The full reference design has been implemented using the C8051F301, and test boards are available for review.

The Si570 I^2C programmable XO is pin and register compatible with the Si598. The Si571 VCXO is pin and register compatible with the Si599. This code allows each device to be interchanged according the requirements of the application. See each device's respective data sheet for frequency range, stability, and jitter performance information.

2. Compiler

This reference design requires the use of single-precision floating point calculations. In order to reduce the cost burden associated with this requirement, the SDCC compiler has been employed. The resulting binary can be downloaded to the C8051F301 Flash memory space. “AN198: Integrating SDCC 8051 Tools into the Silicon Labs IDE” describes the procedure for linking the SDCC compiler to the development environment. The code has also been successfully tested using the full version of the Keil compiler. “AN104: Integrating Keil 8051 Tools into the Silicon Labs IDE” describes the procedure for linking the Keil compiler to the development environment.

3. C8051F301 Specifics

Since I^2C communication is required, the code calls out the I^2C hardware within the MCU. When porting the code to other processors, it is necessary to determine how to enable I^2C communication within that device. The I^2C interrupts and functions within this code will most likely not handle the other processor properly and will likely need to be rewritten. Usually, this code is available from the processor vendor. The I^2C code used in this reference design was copied from “AN141: SMBus Communication for Small Form Factor Device Families” (see EEPROM example).

4. Solution

This reference design implements the necessary reads/writes from/to the Si57x/598/599 and the output divider search (for HS_DIV and N1), and the calculation of the new reference frequency multiplier (RFREQ). There are two included code examples: One that will start from the default startup frequency (located in Appendix A) after powerup or reset and one that will start from the last output frequency (located in Appendix B) after powerup or reset. At powerup or reset, the MCU reads the startup configuration from the Si57x/598/599, then waits for the strobe. Initially, the Si57x/598/599 outputs the startup frequency as defined at the time of order (if using the software in Appendix A) or the last output frequency before powerup or reset (if using the software in Appendix B). Once a strobe is received, the MCU calculates the new configuration and loads it into the Si57x/598/599 over I^2C. Both code examples simply cycle through an array of frequencies listed in the constants section of the code (see Table 1). If other frequencies are needed, simply edit and recompile this code as needed.

Table 1. Hard Coded Reference Design Frequency List in MHz

27.000000	61.440000	74.175824	74.250000	100.000000	106.250000
125.000000	133.333300	155.520000	156.250000	161.132813	164.355469
167.331646	167.410714	172.642300	173.370747	176.095145	176.838163
200.000000	212.500000	368.640000	491.520000	622.080000	625.000000
644.531250	657.421875	666.514286	669.326580	672.162712	690.569196
693.482991	704.380580	707.352650	800.000000	933.120000	1000.000000
1400.000000					
Note: The Si598/599 devices are limited to 525 MHz max frequency.					

Because the C8051F301 only supports single-precision floating point arithmetic, the data sheet-specified procedure has been reorganized in this reference design to limit the loss of accuracy.

Taking the standard equations:

$$f_{out} = \frac{f_{XTAL} \times RFREQ}{HSDIV \times N1}$$

and

$$RFREQ_{NEW} = \frac{f_{DCO}}{f_{XTAL}}$$

then substituting for the crystal frequency in the second equation gives:

$$RFREQ_{NEW} = \left(\frac{f_{OUT1}}{f_{OUT0}}\right)\left(\frac{N1_1}{N1_0}\right)\left(\frac{HSDIV_1}{HSDIV_0}\right) \times RFREQ_{OLD}$$

where the parenthetical calculations can be kept close to unity. By keeping these ratios close to unity, very little precision is lost. If the startup output frequency (fout0) is large or small relative to the new output frequency, it may be necessary to commute the factors with the goal of keeping each division calculation close to unity.

5. Hardware

The PCB schematic shown in Figure 2 is used to demonstrate the code. The MCU transitions to the next frequency in the list following the pressing of a switch (SW1). The status LEDs light in sequence to show the index of the frequency currently selected.

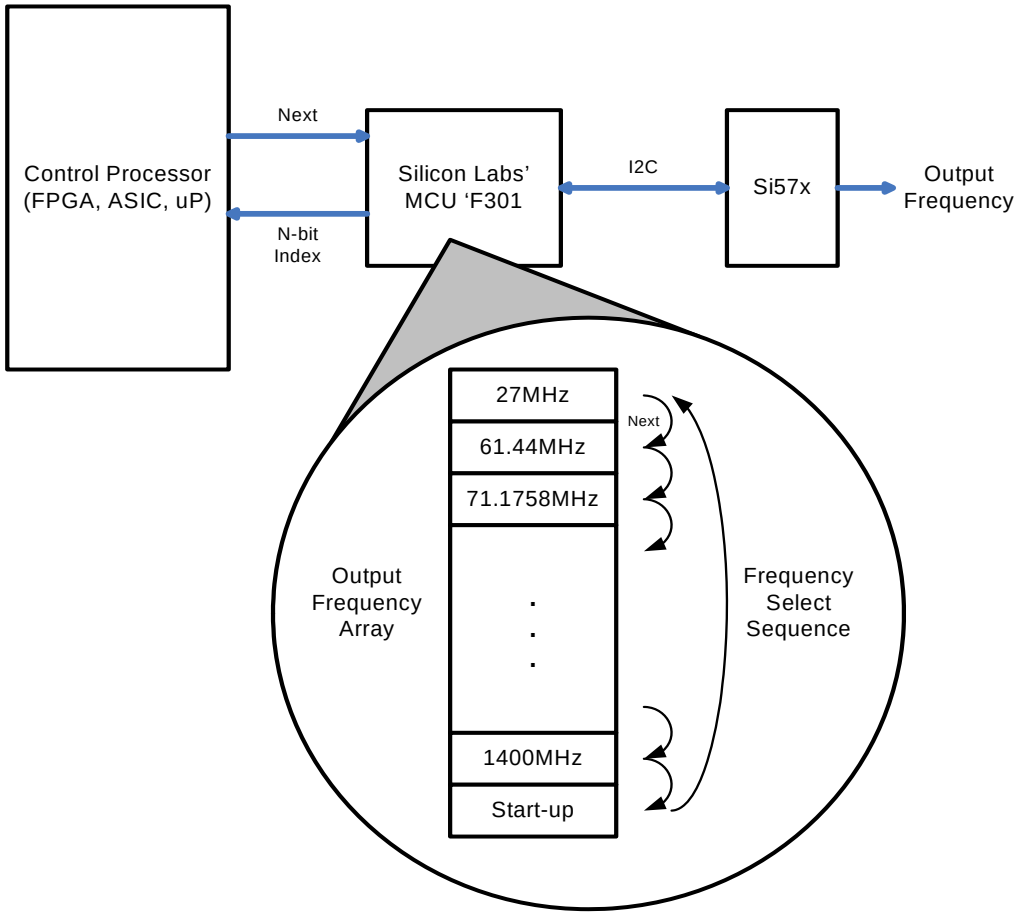

Figure 1. Reference Design Expected Usage Model

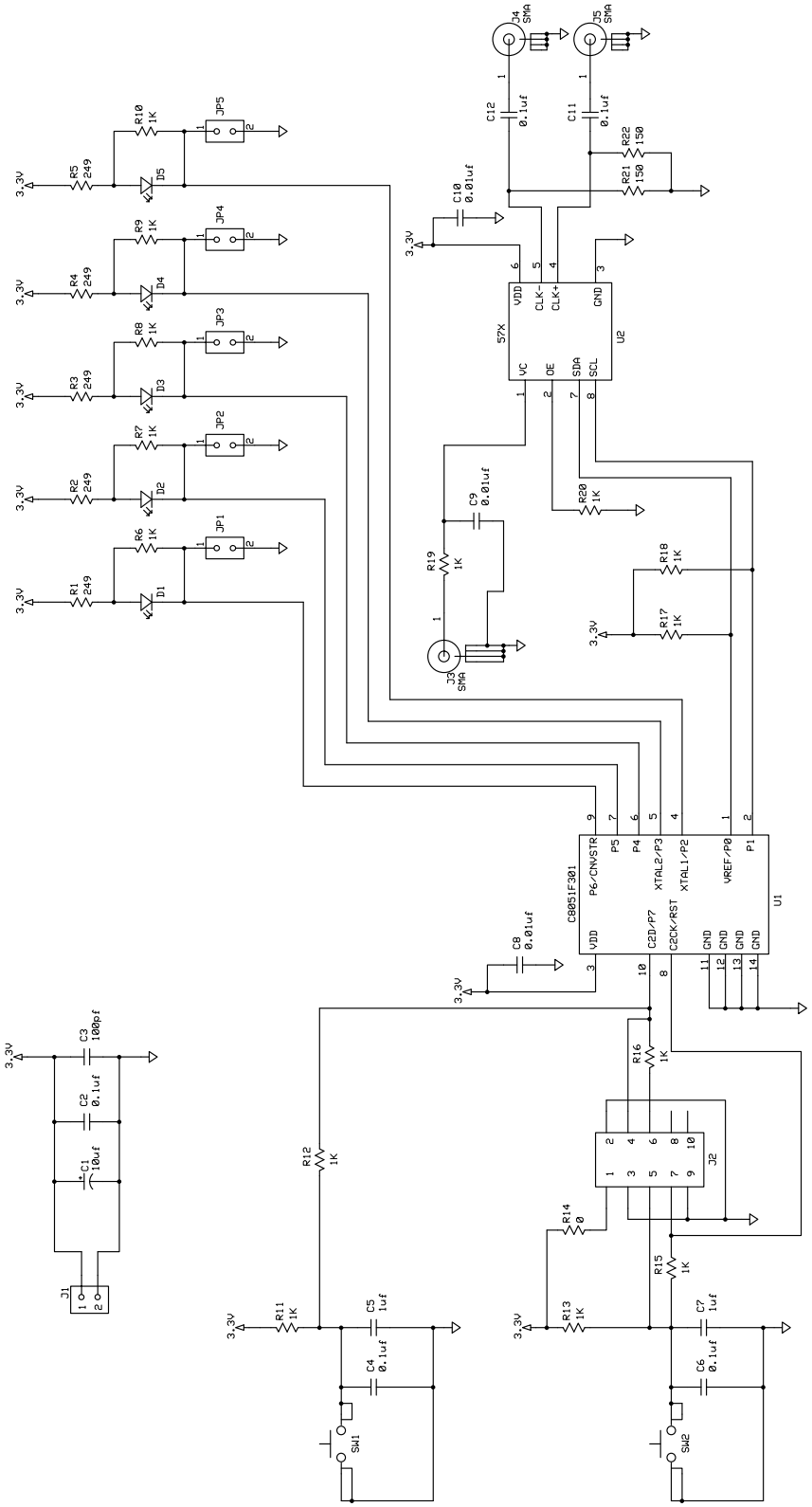

Figure 2. PCB Schematic

6. Conclusion

The calculations necessary for programming new frequencies on the Si570/Si571 in a real system have been demonstrated. The demonstration hardware can be duplicated in any design with very little loss of area. The reference design C code is available for download from Silicon Labs and can be ported to other processor platforms with minimal changes. The precision of the calculations must be ensured to achieve the frequency accuracy demanded by most systems. The precision is met within this code by reorganizing the calculations and using the unsigned long data type. Code within “Appendix A—Reference Design Software” demonstrates the calculations required, and code within “Appendix B—Reference Design Software with Non-volatile Frequency Pointer Storage” adds non-volatile storage of the last frequency selected. The reference design code was tested using the Si570; code for the Si598 may require slight modifications because the Si598's RFREQ value has non-zero bits above the 34th digit.

APPENDIX A—REFERENCE DESIGN SOFTWARE

```
//-----------------------------------------------------------------------------
// Si57x_FreqProgFirmware_F300_A.c
//-----------------------------------------------------------------------------
// Copyright 2008 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This software communicates with a Si570 via the SMBus interface with an 'F30x
// device to program the output frequency. By pressing the 'next' button on the
// board, the next output frequency in FOUT1 will be output. The MCU will always
// start from the default start-up frequency. The software is compatible with the Si598/599.
// When using those devices, the max frequency should be limited to 525 MHz.
//
// Note: This code has been successfully tested using SDCC (ver 2.8.0), Keil
//       (ver 8.0), and Hi-Tech (ver 9.01)
//
// - Interrupt-driven SMBus implementation
// - Only master states defined (no slave or arbitration)
// - Timer1 used as SMBus clock source
// - Timer2 used by SMBus for SCL low timeout detection
// - SCL frequency defined by <SMB_FREQUENCY> constant
// - ARBLOST support included
// - Pinout:
//    P0.0 -> SDA (SMBus)
//    P0.1 -> SCL (SMBus)
//    P0.2 -> LED
//    P0.3 -> LED
//    P0.4 -> LED
//    P0.5 -> LED
//    P0.6 -> LED
//    P0.7 -> 'Next' button on board
//
//    all other port pins unused
//
// How To Test:
//
// 1) Download code to a 'F30x device that is connected to an Si570 device via SMBus.
//   (Note: In this example, SDA is connected to P0.0 and SCL is connected to P0.1)
// 2) Run the code:
//         a) The output of the Si570 will cycle through the predefined list called
//            FOUT1. Each time the 'next' button is pressed, the output frequency will
//            change.
//         b) The frequencies in FOUT1 can be changed, but do not forget to update the
//            global definition FREQ_LIST_LENGTH with the corresponding length of the
//            of the new array.
//
//
// Target:         C8051F30x
// Tool chain:     SDCC 2.8.0
// Command Line:   None
//
// Release 1.0
//    -Initial Revision (ES)
//    -25 AUG 2008
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------
```

```
#include <compiler_defs.h>
#include <c8051F300_defs.h>           // SFR declarations
#include <math.h>                     // Used for the floorf() and ceilf() function

//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------
// floor function is called differently in Keil, SDCC, and Hi-Tech

#if defined SDCC
   # define FLOORF(value)     floorf (value)
   # define CEILF(value)     ceilf (value)
#elif defined __C51__ | HI_TECH_C
   # define FLOORF(value)     floor (value)
   # define CEILF(value)     ceil (value)
#endif

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------

#define SYSCLK            24500000        // System clock frequency in Hz

#define SMB_FREQUENCY     100000          // Target SCL clock rate
                                          // Can be 100kHz or 400kHz

#define POW_2_16          65536.0         // Floating point constants
#define POW_2_24          16777216.0      // used in approach that can handle
#define POW_2_28          268435456.0     // double precision floating point
                                          // calculations

#define WRITE             0x00            // SMBus WRITE command
#define READ              0x01            // SMBus READ command

// Device addresses (7 bits, lsb is a don't care)
#define  SLAVE_ADDR      0xAA             // Device address for slave target

// SMBus Buffer Size
#define  SMB_BUFF_SIZE    0x08            // Defines the maximum number of bytes
                                          // that can be sent or received in a
                                          // single transfer

// Status vector - top 4 bits only
#define  SMB_MTSTA        0xE0            // (MT) start transmitted
#define  SMB_MTDB         0xC0            // (MT) data byte transmitted
#define  SMB_MRDB         0x80            // (MR) data byte received
// End status vector definition

#define FREQ_LIST_LENGTH   38            // length of the frequency list
                                          // (includes start-up frequency)

// Enter in the Startup frequency. This should be known when ordering the chip, but if not,
// this can be measured. This must be accurate to ensure accuracy of all other frequencies.
#define  FOUT_START_UP    100      // MHz

// Si57x/598/599's FDCO Range
// These values should not be modified, but can be depeneding on application.
SEGMENT_VARIABLE(FDCO_MAX, float, SEG_CODE) = 5670;  //MHz
SEGMENT_VARIABLE(FDCO_MIN, float, SEG_CODE) = 4850;  //MHz

// Change the FOUT0 and FOUT1 to configure the Si57x/598/599
// frequency plan. See datasheet for FOUT0 and FOUT1 ranges.
```

```
// Both values must be given in MHz.
SEGMENT_VARIABLE(FOUT0, float, SEG_CODE) = FOUT_START_UP;

// Array of frequencies that can be cycled by pressing 'NEXT' button
// Note: Values written to array must be within range of Si57x/598/599 and in MHz.
// Update Global Constant FREQ_LIST_LENGTH if length of array changes.
SEGMENT_VARIABLE(FOUT1[FREQ_LIST_LENGTH], float, SEG_CODE) =
{27.000000 ,
61.440000 ,
74.175824 ,
74.250000 ,
100.000000 ,
106.250000 ,
125.000000 ,
133.333300 ,
155.520000 ,
156.250000 ,
161.132813 ,
164.355469 ,
167.331646 ,
167.410714 ,
172.642300 ,
173.370747 ,
176.095145 ,
176.838163 ,
200.000000 ,
212.500000 ,
368.640000 ,
491.520000 ,
622.080000 ,
625.000000 ,
644.531250 ,
657.421875 ,
666.514286 ,
669.326580 ,
672.162712 ,
690.569196 ,
693.482991 ,
704.380580 ,
707.352650 ,
800.000000 ,
933.120000 ,
1000.000000 ,
1400.000000 ,
FOUT_START_UP };     // keeps start up frequency in the loop

// These are the only valie values for HS_DIV. See datasheet for more information
SEGMENT_VARIABLE(HS_DIV[6], U8, SEG_CODE) = {11, 9, 7, 6, 5, 4};


//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------

bit NEXT = 0;        // Flag bit that is set ISR after 'next'
                     // button is pressed

U8 INITIAL_HSDIV;    // HSDIV value read from the Si57x/598/599 when it
                     // when it is turned on

U8 INITIAL_N1;       // N1 value read from the Si57x/598/599 when it
                     // when it is turned on

U8 REG[6];           // Array of bits that holds the initial
```

```
                    // values read from the Si57x/598/599

U16 CURR_FREQ;      // Holds the index of the current frequency
                    // being output

U32 INITIAL_RFREQ_LONG;    // RFREQ value read from the Si57x/598/599 when it
                           // when it is turned on

float RFREQ;        // Fractional mulitplier used to achieve
                    // correct output frequency

float FXTAL;        // Will hold the value of the internal crystal
                    // frequency


U8* pSMB_DATA_IN;   // Global pointer for SMBus data
                    // All receive data is written here

U8 SMB_SINGLEBYTE_OUT;          // Global holder for single byte writes.

U8* pSMB_DATA_OUT;              // Global pointer for SMBus data.
                                // All transmit data is read from here

U8 SMB_DATA_LEN;                // Global holder for number of bytes
                                // to send or receive in the current
                                // SMBus transfer.

U8 WORD_ADDR;                   // Global holder for the  word
                                // address that will be accessed in
                                // the next transfer

U8 TARGET;                      // Target SMBus slave address

//U32 FRAC_BITS;                // Double Floating Point Precision Method

volatile bit SMB_BUSY = 0;      // Software flag to indicate when the
                                // I2C_ByteRead() or
                                // I2C_ByteWrite()
                                // functions have claimed the SMBus

bit SMB_RW;                     // Software flag to indicate the
                                // direction of the current transfer

bit SMB_SENDWORDADDR;           // When set, this flag causes the ISR
                                // to send the 8-bit <WORD_ADDR>
                                // after sending the slave address.


bit SMB_RANDOMREAD;             // When set, this flag causes the ISR
                                // to send a START signal after sending
                                // the word address.
                                // The ISR handles this
                                // switchover if the <SMB_RANDOMREAD>
                                // bit is set.

bit SMB_ACKPOLL;                // When set, this flag causes the ISR
                                // to send a repeated START until the
                                // slave has acknowledged its address

SBIT(SDA, SFR_P0, 0);           // SMBus on P0.0
SBIT(SCL, SFR_P0, 1);           // and P0.1

//-----------------------------------------------------------------------------
// Function PROTOTYPES
```

```
//-----------------------------------------------------------------------------

void SMBus_Init (void);
void Timer1_Init (void);
void Timer2_Init (void);
void Int0_Init (void);
void Port_Init (void);
INTERRUPT_PROTO(SMBus_ISR, INTERRUPT_SMBUS0);
INTERRUPT_PROTO(ButtonPushed_ISR, INTERRUPT_INT0);
INTERRUPT_PROTO(Timer2_ISR, INTERRUPT_TIMER2);
void RunFreqProg (void);
U8 SetBits (U8 original, U8 reset_mask, U8 new_val);
void ReadStartUpConfiguration (void);
void UpdateStatusOutput (void);
void I2C_ByteWrite (U8 addr, U8 dat);
U8 I2C_ByteRead (U8 addr);


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
//
// Main routine performs all configuration tasks, then loops forever waiting
// for the 'next' button on the board to be pressed. When pressed, the Si570
// will then output the next frequency.
//
void main (void){
   U8 i;                               // Temporary counter variable used in
                                       // for loops
   PCA0MD &= ~0x40;                    // WDTE = 0 (disable watchdog timer)
   OSCICN |= 0x03;                     // Configure internal oscillator for
                                       // its maximum frequency (24.5 Mhz)

   // If slave is holding SDA low because of an improper SMBus reset or error
   while(!SDA)
   {
      // Provide clock pulses to allow the slave to advance out
      // of its current state. This will allow it to release SDA.
      XBR1 = 0x40;                     // Enable Crossbar
      SCL = 0;                         // Drive the clock low
      for(i = 0; i < 255; i++);        // Hold the clock low
      SCL = 1;                         // Release the clock
      while(!SCL);                     // Wait for open-drain
                                       // clock output to rise
      for(i = 0; i < 10; i++);         // Hold the clock high
      XBR1 = 0x00;                     // Disable Crossbar
   }

   Port_Init ();                       // Initialize Crossbar and GPIO

   CKCON = 0x10;                       // Timer 1 is sysclk
                                       // Timer 2 is sysclk/12 (see TMR2CN)

   Timer1_Init ();                     // Configure Timer1 for use as SMBus
                                       // clock source

   Timer2_Init ();                     // Configure Timer2 for use with SMBus
                                       // low timeout detect

   SMBus_Init ();                      // Configure and enable SMBus

   Int0_Init ();                       // Configure INT0

   EIE1  |= 0x01;           // Enable SMBus interrupt
```

```
   EX0    = 1;              // Enable External Interrupt 0
   EA     = 1;              // Global interrupt enable  ****MUST BE LAST****

   IE0 = 0;                 // Clears INT0 interrupt flag in case previously set

   CURR_FREQ = FREQ_LIST_LENGTH - 1;   // Start up frequency is the last element in the array.
                                       // Set CURR_FREQ index to point to last element

   ReadStartUpConfiguration();

   UpdateStatusOutput();          // Update LED's based on which freqency in array FOUT1 is
                                  // being output

   CURR_FREQ = 0;           // After 'next' is pressed, first frequency element in FOUT1
                            // will be output

   while(1)                 // Loop forever
   {
      while (NEXT == 0) {}        // Wait for button to be pressed
      RunFreqProg();
     NEXT = 0;                    // Clear NEXT variable
     UpdateStatusOutput();        // Update LEDs on the board
     CURR_FREQ = CURR_FREQ + 1;   // Update index for next frequency
     if (CURR_FREQ > (FREQ_LIST_LENGTH - 1))
     {
        CURR_FREQ = 0;            // Wrap around index after overflow
      }
   }
}


//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SMBus_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// The SMBus peripheral is configured as follows:
// - SMBus enabled
// - Slave mode disabled
// - Timer1 used as clock source. The maximum SCL frequency will be
//   approximately 1/3 the Timer1 overflow rate
// - Setup and hold time extensions enabled
// - Free and SCL low timeout detection enabled
//
void SMBus_Init (void)
{
   SMB0CF = 0x5D;                      // Use Timer1 overflows as SMBus clock
                                       // source;
                                       // Disable slave mode;
                                       // Enable setup & hold time extensions;
                                       // Enable SMBus Free timeout detect;
                                       // Enable SCL low timeout detect;

   SMB0CF |= 0x80;                     // Enable SMBus;
}


//-----------------------------------------------------------------------------
```

```
// Timer1_Init
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Timer1 is configured as the SMBus clock source as follows:
// - Timer1 in 8-bit auto-reload mode
// - SYSCLK / 12 as Timer1 clock source
// - Timer1 overflow rate => 3 * SMB_FREQUENCY
// - The maximum SCL clock rate will be ~1/3 the Timer1 overflow rate
// - Timer1 enabled
//
void Timer1_Init (void)
{
// Make sure the Timer can produce the appropriate frequency in 8-bit mode
// Supported SMBus Frequencies range from 10kHz to 100kHz.  The CKCON register
// settings may need to change for frequencies outside this range.
   TMOD = 0x20;                              // Timer1 in 8-bit auto-reload mode
   TH1 = 0xFF - (SYSCLK/SMB_FREQUENCY/3) + 1; // 100kHz or 400kHz for SCL
   TL1 = TH1;                                // Init Timer1
   TR1 = 1;                                  // Timer1 enabled
}


//-------------------------------------------------------------------------
// Timer2_Init
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Timer2 configured for Si57x/598/599 reset delay as
// follows:
// - Timer2 in 16-bit auto-reload mode
// - SYSCLK/12 as Timer2 clock source
// - Timer2 reload registers loaded for RESET_DELAY_TIME overflow
// - Timer2 pre-loaded to overflow after RESET_DELAY_TIME
//
void Timer2_Init (void)
{
   TMR2CN = 0x00;                     // Timer2 configured for 16-bit auto-
                                      // reload, low-byte interrupt disabled
                                      // Timer2 uses SYSCLK/12 (see CKCON)

   TMR2RLL = 0x5F;
   TMR2RLH = 0x88;
   TMR2L = TMR2RLL;
   TMR2H = TMR2RLH;
   TF2LEN = 0;
   TF2H = 0;
   TF2L = 0;
   ET2  = 1;                          // Timer2 interrupt enable
   TR2 = 1;                           // Start Timer2
}


//-------------------------------------------------------------------------
// Int0_Init
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
```

```
// Configure INT0 to trigger on P0.7
//
void Int0_Init (void)
{
   IT01CF    |= 0x07;
   PX0  = 1;               // Set External Interrupt 0 to high priority
   IT0  = 1;               // Configure External Interrupt to be edge
}                          // triggered


//-----------------------------------------------------------------------------
// Port_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Configure the Crossbar and GPIO ports.
//
// P0.0   digital   open-drain    SMBus SDA
// P0.1   digital   open-drain    SMBus SCL
// P0.2   digital   open-drain    LED on board to show frequency index
// P0.3   digital   open-drain    LED on board to show frequency index
// P0.4   digital   open-drain    LED on board to show frequency index
// P0.5   digital   open-drain    LED on board to show frequency index
// P0.6   digital   open-drain    LED on board to show frequency index
// P0.7   digital   open-drain    Used as input switch 'next' on board
//
// all other port pins unused
//
// Note: If the SMBus is moved, the SCL and SDA sbit declarations must also
// be adjusted.
//
void Port_Init (void)
{
   P0MDOUT = 0x00;                    // All P0 pins open-drain output

   XBR1 = 0x04;                       // Enable SMBus
   XBR2 = 0x40;                       // Enable crossbar and weak pull-ups
}


//-----------------------------------------------------------------------------
// SMBus Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// SMBus ISR state machine
// - Master only implementation - no slave or arbitration states defined
// - All incoming data is written starting at the global pointer <pSMB_DATA_IN>
// - All outgoing data is read from the global pointer <pSMB_DATA_OUT>
//
INTERRUPT(SMBus_ISR, INTERRUPT_SMBUS0)
{
   bit FAIL = 0;                      // Used by the ISR to flag failed
                                      // transfers

   static char i;                     // Used by the ISR to count the
                                      // number of data bytes sent or
                                      // received

   static bit SEND_START = 0;         // Send a start

   switch (SMB0CN & 0xF0)             // Status vector
   {
```

```
// Master Transmitter/Receiver: START condition transmitted.
case SMB_MTSTA:
   SMB0DAT = TARGET;             // Load address of the target slave
   SMB0DAT &= 0xFE;              // Clear the LSB of the address for the
                                 // R/W bit
   SMB0DAT |= SMB_RW;            // Load R/W bit
   STA = 0;                      // Manually clear START bit
   i = 0;                        // Reset data byte counter
   break;

// Master Transmitter: Data byte (or Slave Address) transmitted
case SMB_MTDB:
   if (ACK)                      // Slave Address or Data Byte
   {                             // Acknowledged?
      if (SEND_START)
      {
         STA = 1;
         SEND_START = 0;
         break;
      }
      if(SMB_SENDWORDADDR)       // Are we sending the word address?
      {
         SMB_SENDWORDADDR = 0;   // Clear flag
         SMB0DAT = WORD_ADDR;    // Send word address

         if (SMB_RANDOMREAD)
         {
            SEND_START = 1;      // Send a START after the next ACK cycle
            SMB_RW = READ;
         }

         break;
      }

      if (SMB_RW==WRITE)         // Is this transfer a WRITE?
      {

         if (i < SMB_DATA_LEN)   // Is there data to send?
         {
            // send data byte
            SMB0DAT = *pSMB_DATA_OUT;

            // increment data out pointer
            pSMB_DATA_OUT++;

            // increment number of bytes sent
            i++;
         }
         else
         {
           STO = 1;              // Set STO to terminte transfer
           SMB_BUSY = 0;         // Clear software busy flag
         }
      }
      else {}                    // If this transfer is a READ,
                                 // then take no action. Slave
                                 // address was transmitted. A
                                 // separate 'case' is defined
                                 // for data byte recieved.
   }
   else                          // If slave NACK,
   {
      if(SMB_ACKPOLL)
      {
```

```
               STA = 1;                // Restart transfer
            }
            else
            {
               FAIL = 1;               // Indicate failed transfer
            }                          // and handle at end of ISR
         }
         break;

      // Master Receiver: byte received
      case SMB_MRDB:
         if ( i < SMB_DATA_LEN )       // Is there any data remaining?
         {
            *pSMB_DATA_IN = SMB0DAT;   // Store received byte
            pSMB_DATA_IN++;            // Increment data in pointer
            i++;                       // Increment number of bytes received
            ACK = 1;                   // Set ACK bit (may be cleared later
                                       // in the code)

         }

         if (i == SMB_DATA_LEN)        // This is the last byte
         {
            SMB_BUSY = 0;              // Free SMBus interface
            ACK = 0;                   // Send NACK to indicate last byte
                                       // of this transfer
            STO = 1;                   // Send STOP to terminate transfer
         }

         break;

      default:
         FAIL = 1;                     // Indicate failed transfer
                                       // and handle at end of ISR
         break;
   }

   if (FAIL)                           // If the transfer failed,
   {
      SMB0CF &= ~0x80;                 // Reset communication
      SMB0CF |= 0x80;
      STA = 0;
      STO = 0;
      ACK = 0;

      SMB_BUSY = 0;                    // Free SMBus

      FAIL = 0;
   }

   SI = 0;                             // Clear interrupt flag
}


//-----------------------------------------------------------------------------
// External Interrupt 0 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// An external interrupt 0 indicates that the 'next' button on the board
// has been pressed. The ISR sets, 'next' to 1 to allow new frequency to be set
//
INTERRUPT (ButtonPushed_ISR, INTERRUPT_INT0)
{
  NEXT = 1;
```

```
}


//-----------------------------------------------------------------------------
// Timer2 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// A Timer2 interrupt indicates an SMBus SCL low timeout.
// The SMBus is disabled and re-enabled if a timeout occurs.
//
INTERRUPT(Timer2_ISR, INTERRUPT_TIMER2)
{
   SMB0CF &= ~0x80;                    // Disable SMBus
   SMB0CF |= 0x80;                     // Re-enable SMBus
   TF2H = 0;                           // Clear Timer2 interrupt-pending flag
   SMB_BUSY = 0;                       // Free bus
}


//-----------------------------------------------------------------------------
// Support Functions
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// RunFreqProg
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Program Si570 to output the next frequency in the list FOUT1
//
void RunFreqProg (void)
{
   U8 i;                    // Temporary counter variable used in for loops

   U8 n1;                   // Output divider that is modified and used in
                            // calculating the new RFREQ

   U8 hsdiv;                // Output divider that is modified and used in
                            // calculating the new RFREQ

   bit validCombo;          // Flag that is set to 1 if a valid combination
                            // of N1 and HS_DIV is found

   U8 reg137;               // Stores the contents of Register 137 on Si57x/598/599

   U16 divider_max;         // Maximum divider for HS_DIV and N1 combination

   U16 curr_div;            // Minimum divider for HS_DIV and N1 combination

//  U16 whole;               // Used in Double Floating Point Precision Method

   U32 final_rfreq_long;    // Final REFREQ that is sent to the Si57x/598/599

        float curr_n1;      // Used to calculate the final N1 to send to the
        float n1_tmp;       // Si570

   float ratio = 0;         // Will hold the final ratio to multiply the initial
                            // REFREQ by to acheive the final RFREQ

   // Find dividers (get the max and min divider range for the HS_DIV and N1 combo)
   divider_max = FLOORF(FDCO_MAX / FOUT1[CURR_FREQ]);
   curr_div = CEILF(FDCO_MIN / FOUT1[CURR_FREQ]);
```

```
validCombo = 0;

while (curr_div <= divider_max)
{
   //check all the HS_DIV values with the next curr_div
   for(i=0; i<6; i++)
   {
      // get the next possible n1 value
      hsdiv = HS_DIV[i];
      curr_n1 = (float)(curr_div) / (float)(hsdiv);

      // Determine if curr_n1 is an integer and an even number or one
      // then it will be a valid divider option for the new frequency
      n1_tmp = FLOORF(curr_n1);
      n1_tmp = curr_n1 - n1_tmp;
      if(n1_tmp == 0.0)  // Then curr_n1 is an integer
      {
         n1 = (U8) curr_n1;

         if( (n1 == 1) || ((n1 & 1) == 0) ) // Then the calculated N1 is
         {                                  // either 1 or an even number
            validCombo = 1;
         }
      }
      if(validCombo == 1) break;       // Divider was found, exit loop
   }
   if(validCombo == 1) break;          // Divider was found, exit loop

   curr_div = curr_div + 1;            // If a valid divider is not found,
                                       // increment curr_div and loop
}

// If validCombo == 0 at this point, then there is an error
// in the calculation. Check if the provided FOUT0 and FOUT1
// are valid frequencies

// New RFREQ calculation
RFREQ = (FOUT1[CURR_FREQ] * n1 * hsdiv) / FXTAL;

// Calculate RFREQ organizing the float variables to save precision;
// RFREQ is kept as an unsigned long
// only 32 bits are available in the long format
// RFREQ in the device has 34 bits of precision
// only 34 of the 38 bits are needed since RFREQ is between 42.0 and
// 50.0 for fxtal of 114.285MHz (nominal)

ratio = FOUT1[CURR_FREQ] / FOUT0;           // Try to keep ration near 1
                                            // to maintain precision
ratio = ratio * (((float)n1)/((float)INITIAL_N1));
ratio = ratio * (((float)hsdiv)/((float)INITIAL_HSDIV));
final_rfreq_long = ratio * INITIAL_RFREQ_LONG;  // Calculate final RFREQ
                                                // value using ratio
                                                // computed above

for(i = 0; i < 6; i++)
{
   REG[i] = 0;               //clear registers
}

hsdiv = hsdiv - 4;  // Subtract 4 because of the offset of HS_DIV.
                    // Ex: "000" maps to 4, "001" maps to 5

//set the top 3 bits of REG[0] which will correspond to Register 7 on Si57x/598/599
REG[0] = (hsdiv << 5);
```

```
   // convert new N1 to the binary representation
   if(n1 == 1)
   {
      n1 = 0;  //Corner case for N1. If N1=1, it is represented as "00000000"
   }
   else if((n1 & 1) == 0)
   {
      n1 = n1 - 1; // If n1 is even, round down to closest odd number. See the
   }               // Si57x/598/599 data sheet for more information.

   // Write correct new values to REG[0] through REG[6]
   // These will be sent to the Si57x/598/599 and will update the output frequency
   REG[0] = SetBits(REG[0], 0xE0, (n1 >> 2)); // Set N1 part of REG[0]
   REG[1] = (n1 & 3) << 6;                    // Set N1 part of REG[1]
   //Write new version of RFREQ to corresponding registers
   REG[1] = REG[1] | (final_rfreq_long >> 30);
   REG[2] = final_rfreq_long >> 22;
   REG[3] = final_rfreq_long >> 14;
   REG[4] = final_rfreq_long >> 6;
   REG[5] = final_rfreq_long << 2;

/*
   // Double Floating Point Precision Method
   // convert new RFREQ to the binary representation
   // separate the integer part
   whole = FLOORF(RFREQ);

   // get the binary representation of the fractional part
   FRAC_BITS = FLOORF((RFREQ - whole) * POW_2_28);

   // set reg 12 to 10 making frac_bits smaller by
   // shifting off the last 8 bits everytime
   for(i=5; i >=3; i--)
   {
      REG[i] = FRAC_BITS & 0xFF;
      FRAC_BITS = FRAC_BITS >> 8;
   }
   // set the last 4 bits of the fractional portion in reg 9
   REG[2] = SetBits(REG[2], 0xF0, (FRAC_BITS & 0xF));

   // set the integer portion of RFREQ across reg 8 and 9
   REG[2] = SetBits(REG[2], 0x0F, (whole & 0xF) << 4);
   REG[1] = SetBits(REG[1], 0xC0, (whole >> 4) & 0x3F);
*/

   reg137 = I2C_ByteRead(137);            // Read the current state of Register 137

   I2C_ByteWrite(137, reg137 | 0x10);     // Set the Freeze DCO bit in that register
                                          // This must be done in order to update
                                          // Registers 7-12 on the Si57x/598/599
   for(i=0; i<6; i++)
   {
      I2C_ByteWrite(i+7, REG[i]);         // Write the new values to Registers 7-12
   }

   reg137 = I2C_ByteRead(137);            // Read the current state of Register 137

   I2C_ByteWrite(137, reg137 & 0xEF);     // Clear the Freeze DCO bit

   I2C_ByteWrite(135, 0x40);              // Set the NewFreq bit to alert the DPSLL
                                          // that a new frequency configuration
                                          // has been applied
```

```
}


//-----------------------------------------------------------------------------
// SetBits
//-----------------------------------------------------------------------------
//
// Return Value : unsigned char
// Parameters   :
//   1) unsigned char original - original state of variable to be changed
//
//   2) unsigned char reset_mask - contains mask of bits that will reset the
//      the original char variable.
//
//   3) unsigned char new_val - contains a mask of bits that need to be set in
//      the original variable.
//
// This function sets appropriate bits in an unsigned char variable
//
U8 SetBits(U8 original, U8 reset_mask, U8 new_val)
{
   return (( original & reset_mask ) | new_val );
}


//-----------------------------------------------------------------------------
// ReadStartUpConfig
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Reads start-up register contents for RFREQ, HS_DIV, and N1 and calculates
// the internal crystal frequency (FXTAL)
//
void ReadStartUpConfiguration (void)
{
   U8 i;                        // Counter used in for loops
   I2C_ByteWrite(135, 0x01);  // Writes 0x01 to register 135. This
                              // will recall NVM bits into RAM. See
                              // register 135 in Si57x/598/599 datasheet for
                              // more information

   // read registers 7 to 12 of Si57x/598/599
   // REG[0] is equivalent to register 7 in the device
   // REG[5] is register 12 in the device
   for(i=0; i<6; i++)
   {
      REG[i] = I2C_ByteRead(i+7);
   }

   INITIAL_HSDIV = ((REG[0] & 0xE0) >> 5) + 4; // Get value fo INITIAL_HSDIV from REG[0]
   // 4 is added to this because the bits "000" correspond to an HSDIV of 4, so there is
   // an offset of 4. See the register 7 of Si570 datasheet for more information.

   INITIAL_N1 = (( REG[0] & 0x1F ) << 2 ) + (( REG[1] & 0xC0 ) >> 6 );
   // Get correct value of INITIAL_N1 by adding parts of REG[0] and REG[1]

   if(INITIAL_N1 == 0)
   {
      INITIAL_N1 = 1;              // This is a corner case of N1
   }
   else if(INITIAL_N1 & 1 != 0)
   {
```

```
      INITIAL_N1 = INITIAL_N1 + 1; // As per datasheet, illegal odd divider values should
                                   // should be rounded up to the nearest even value.
   }

   // Double Floating Point Precision Method
   // RFREQ conversion (reconstruct the fractional portion (bits 0 to 28) from the registers)
   // (this method requires double precision floating point data type to be accurate)
/* FRAC_BITS = (( REG[2] & 0xF ) * POW_2_24 );
   FRAC_BITS = FRAC_BITS + (REG[3] * POW_2_16);
   FRAC_BITS = FRAC_BITS + (REG[4] * 256);
   FRAC_BITS = FRAC_BITS + REG[5];

   RFREQ = FRAC_BITS;
   RFREQ = RFREQ / POW_2_28;
*/

   // Read initial value for RFREQ. A 34-bit number is fit into a 32-bit space by
   // ignoring lower 2 bits.
   INITIAL_RFREQ_LONG = ( REG[1] & 0x3F );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[2] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[3] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[4] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 6) + ( REG[5] >> 2 );

   // RFREQ conversion (reconstruct the integer portion from the registers)
   RFREQ = RFREQ + ( (( REG[1] & 0x3F ) << 4 ) + (( REG[2] & 0xF0 ) >> 4 ) );

   // Crystal Frequency (FXTAL) calculation
   FXTAL = (FOUT0 * INITIAL_N1 * INITIAL_HSDIV) / RFREQ;          //MHz
}


//-----------------------------------------------------------------------------
// UpdateStatusOutput
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// This function updates status of the LED's on the board by reading the
// the variable currFreq and updating P0. The LED's use a binary counting
// system. Writing a 0 to the corresponding pin in Port 0 will light up LED
//
void UpdateStatusOutput (void)
{
   P0 = (0x80 | ((~CURR_FREQ) << 2));
}


//-----------------------------------------------------------------------------
// I2C_ByteWrite
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1) unsigned char addr - address to write in the device via I2C
//      range is full range of character: 0 to 255
//
//   2) unsigned char dat - data to write to the address <addr> in the device
//      range is full range of character: 0 to 255
//
// This function writes the value in <dat> to location <addr> in the device
// then polls the device until the write is complete.
//
```

```
void I2C_ByteWrite (U8 addr, U8 dat)
{
   while (SMB_BUSY);                   // Wait for SMBus to be free.
   SMB_BUSY = 1;                       // Claim SMBus (set to busy)

   // Set SMBus ISR parameters
   TARGET = SLAVE_ADDR;                // Set target slave address
   SMB_RW = WRITE;                     // Mark next transfer as a write
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave Address
   SMB_RANDOMREAD = 0;                 // Do not send a START signal after
                                       // the word address
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling (The ISR
                                       // will automatically restart the
                                       // transfer if the slave does not
                                       // acknoledge its address.

   // Specify the Outgoing Data
   WORD_ADDR = addr;                   // Set the target address in the
                                       // device's internal memory space

   SMB_SINGLEBYTE_OUT = dat;           // Store <dat> (local variable) in a
                                       // global variable so the ISR can read
                                       // it after this function exits

   // The outgoing data pointer points to the <dat> variable
   pSMB_DATA_OUT = &SMB_SINGLEBYTE_OUT;

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte

   // Initiate SMBus Transfer
   STA = 1;
}


//-----------------------------------------------------------------------------
// I2C_ByteRead
//-----------------------------------------------------------------------------
//
// Return Value :
//   1) unsigned char data - data read from address <addr> in the device
//      range is full range of character: 0 to 255
//
// Parameters   :
//   1) unsigned char addr - address to read data from the device
//      range is full range of character: 0 to 255
//
// This function returns a single byte from location <addr> in the device then
// polls the <SMB_BUSY> flag until the read is complete.
//
U8 I2C_ByteRead (U8 addr)
{
   U8 return_val;                      // Holds the return value

   while (SMB_BUSY);                   // Wait for SMBus to be free.
   SMB_BUSY = 1;                       // Claim SMBus (set to busy)

   // Set SMBus ISR parameters
   TARGET = SLAVE_ADDR;                // Set target slave address
   SMB_RW = WRITE;                     // A random read starts as a write
                                       // then changes to a read after
                                       // the repeated start is sent. The
                                       // ISR handles this switchover if
                                       // the <SMB_RANDOMREAD> bit is set.
```

```
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave Address
   SMB_RANDOMREAD = 1;                 // Send a START after the word address
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling


   // Specify the Incoming Data
   WORD_ADDR = addr;                   // Set the target address in the
                                       // devices's internal memory space

   pSMB_DATA_IN = &return_val;         // The incoming data pointer points to
                                       // the <retval> variable.

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte

   // Initiate SMBus Transfer
   STA = 1;
   while(SMB_BUSY);                    // Wait until data is read

   return return_val;

}
```

APPENDIX B—REFERENCE DESIGN SOFTWARE WITH NON-VOLATILE FREQUENCY POINTER STORAGE

```
//-----------------------------------------------------------------------------
// Si57x_FreqProgFirmware_F300_B.c
//-----------------------------------------------------------------------------
// Copyright 2008 Silicon Laboratories, Inc.
// http://www.silabs.com
//
// Program Description:
//
// This software communicates with a Si570 via the SMBus interface with an 'F30x
// device to program the output frequency. By pressing the 'next' button on the
// board, the next output frequency in FOUT1 will be output. The MCU will keep
// track of the current frequency by writing to flash. If the MCU loses power,
// it will restore itself to the last frequency after powering up.
//
// Note: After downloading the code to the MCU and pressing the 'next' button,
//       the last frequency will be stored in flash. To start over and begin
//       outputting the default frequency, you must fill flash with 0xFF.
//       This can be done by going to Tools -> Memory Fill -> Code. After
//       filling the code space with 0xFF, download the code again and then run
//       the code.
//
//       This code has been successfully tested using SDCC (ver 2.8.0), Keil
//       (ver 8.0), and Hi-Tech (ver 9.01)
//.
// - Interrupt-driven SMBus implementation
// - Only master states defined (no slave or arbitration)
// - Timer1 used as SMBus clock source
// - Timer2 used by SMBus for SCL low timeout detection
// - SCL frequency defined by <SMB_FREQUENCY> constant
// - ARBLOST support included
// - Pinout:
//    P0.0 -> SDA (SMBus)
//    P0.1 -> SCL (SMBus)
//    P0.2 -> LED
//    P0.3 -> LED
//    P0.4 -> LED
//    P0.5 -> LED
//    P0.6 -> LED
//    P0.7 -> 'Next' button on board
//
//    all other port pins unused
//
// How To Test:
//
// 1) Download code to a 'F30x device that is connected to an Si570 device via SMBus.
//   (Note: In this example, SDA is connected to P0.0 and SCL is connected to P0.1)
// 2) Run the code:
//         a) The output of the Si570 will cycle through the predefined list called
//            FOUT1. Each time the 'next' button is pressed, the output frequency will
//            change.
//         b) The frequencies in FOUT1 can be changed, but do not forget to update the
//            global definition FREQ_LIST_LENGTH with the corresponding length of the
//            of the new array.
// 3) To start over, fill code space with 0xFF. This will reset the counter in flash
//    will start cycling through the array of frequencies starting with the default
//    start-up frequency.
//
//
// Target:         C8051F30x
// Tool chain:     SDCC 2.8.0
```

```
// Command Line:   None
//
// Release 1.0
//    -Initial Revision (ES)
//    -25 AUG 2008
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------


#include <compiler_defs.h>
#include <c8051F300_defs.h>           // SFR declarations
#include <math.h>                     // Used for the floorf() and ceilf() function

//-----------------------------------------------------------------------------
// Macros
//-----------------------------------------------------------------------------
// floor function is called differently in Keil, SDCC, and Hi-Tech

#if defined SDCC
   # define FLOORF(value)     floorf (value)
   # define CEILF(value)     ceilf (value)
#elif defined __C51__ | HI_TECH_C
   # define FLOORF(value)     floor (value)
   # define CEILF(value)     ceil (value)
#endif

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------

#define  SYSCLK            24500000       // System clock frequency in Hz

#define  SMB_FREQUENCY     100000         // Target SCL clock rate
                                          // Can be 100kHz or 400kHz

#define POW_2_16          65536.0         // Floating point constants
#define POW_2_24          16777216.0      // used in approach that can handle
#define POW_2_28          268435456.0     // double precision floating point
                                          // calculations

#define  WRITE             0x00           // SMBus WRITE command
#define  READ              0x01           // SMBus READ command

// Device addresses (7 bits, lsb is a don't care)
#define  SLAVE_ADDR      0xAA             // Device address for slave target

// SMBus Buffer Size
#define  SMB_BUFF_SIZE    0x08            // Defines the maximum number of bytes
                                          // that can be sent or received in a
                                          // single transfer

// Status vector - top 4 bits only
#define  SMB_MTSTA        0xE0            // (MT) start transmitted
#define  SMB_MTDB         0xC0            // (MT) data byte transmitted
#define  SMB_MRDB         0x80            // (MR) data byte received
// End status vector definition

#define  FREQ_LIST_LENGTH  38    // Length of the frequency list (includes start-up frequency)

// Enter in the Startup frequency. This should be known when ordering the chip, but if not,
// this can be measured. This must be accurate to ensure accuracy of all other frequencies.
```

```
#define  FOUT_START_UP    100             // MHz


#define  FLASH_MEM_ADDR    0x1AAA         // Statically assigned address to write CURR_FREQ
                                          // into Flash

// Si57x/598/599's FDCO Range
// These values should not be modified, but can be depeneding on application.
SEGMENT_VARIABLE(FDCO_MAX, float, SEG_CODE) = 5670;  //MHz
SEGMENT_VARIABLE(FDCO_MIN, float, SEG_CODE) = 4850;  //MHz

// Change the FOUT0 and FOUT1 to configure the Si57x/598/599
// frequency plan. See datasheet for FOUT0 and FOUT1 ranges.
// Both values must be given in MHz.
SEGMENT_VARIABLE(FOUT0, float, SEG_CODE) = FOUT_START_UP;

// Array of frequencies that can be cycled by pressing 'NEXT' button
// Note: Values written to array must be within range of Si57x/598/599 and in MHz.
// Update Global Constant FREQ_LIST_LENGTH if length of array changes.
SEGMENT_VARIABLE(FOUT1[FREQ_LIST_LENGTH], float, SEG_CODE) =
{27.000000 ,
61.440000 ,
74.175824 ,
74.250000 ,
100.000000 ,
106.250000 ,
125.000000 ,
133.333300 ,
155.520000 ,
156.250000 ,
161.132813 ,
164.355469 ,
167.331646 ,
167.410714 ,
172.642300 ,
173.370747 ,
176.095145 ,
176.838163 ,
200.000000 ,
212.500000 ,
368.640000 ,
491.520000 ,
622.080000 ,
625.000000 ,
644.531250 ,
657.421875 ,
666.514286 ,
669.326580 ,
672.162712 ,
690.569196 ,
693.482991 ,
704.380580 ,
707.352650 ,
800.000000 ,
933.120000 ,
1000.000000 ,
1400.000000 ,
FOUT_START_UP };                        // keeps start up frequency in the loop

// These are the only valie values for HS_DIV. See datasheet for more information
SEGMENT_VARIABLE(HS_DIV[6], U8, SEG_CODE) = {11, 9, 7, 6, 5, 4};


//-----------------------------------------------------------------------------
```

```
// Global VARIABLES
//-----------------------------------------------------------------------------

bit NEXT = 0;                   // Flag bit that is set ISR after 'next'
                                // button is pressed

U8 INITIAL_HSDIV;               // HSDIV value read from the Si57x/598/599 when it
                                // when it is turned on

U8 INITIAL_N1;                  // N1 value read from the Si57x/598/599 when it
                                // when it is turned on

U8 REG[6];                      // Array of bits that holds the initial
                                // values read from the Si57x/598/599

U16 CURR_FREQ;                  // Holds the index of the current frequency
                                // being output

U32 INITIAL_RFREQ_LONG;         // RFREQ value read from the Si57x/598/599 when it
                                // when it is turned on

float RFREQ;                    // Fractional mulitplier used to achieve
                                // correct output frequency

float FXTAL;                    // Will hold the value of the internal crystal
                                // frequency


U8* pSMB_DATA_IN;               // Global pointer for SMBus data
                                // All receive data is written here

U8 SMB_SINGLEBYTE_OUT;          // Global holder for single byte writes.

U8* pSMB_DATA_OUT;              // Global pointer for SMBus data.
                                // All transmit data is read from here

U8 SMB_DATA_LEN;                // Global holder for number of bytes
                                // to send or receive in the current
                                // SMBus transfer.

U8 WORD_ADDR;                   // Global holder for the  word
                                // address that will be accessed in
                                // the next transfer

U8 TARGET;                      // Target SMBus slave address

//U32 FRAC_BITS;                // Double Floating Point Precision Method

volatile bit SMB_BUSY = 0;      // Software flag to indicate when the
                                // I2C_ByteRead() or
                                // I2C_ByteWrite()
                                // functions have claimed the SMBus

bit SMB_RW;                     // Software flag to indicate the
                                // direction of the current transfer

bit SMB_SENDWORDADDR;           // When set, this flag causes the ISR
                                // to send the 8-bit <WORD_ADDR>
                                // after sending the slave address.


bit SMB_RANDOMREAD;             // When set, this flag causes the ISR
                                // to send a START signal after sending
                                // the word address.
```

```
                                // The ISR handles this
                                // switchover if the <SMB_RANDOMREAD>
                                // bit is set.

bit SMB_ACKPOLL;                // When set, this flag causes the ISR
                                // to send a repeated START until the
                                // slave has acknowledged its address

SBIT(SDA, SFR_P0, 0);           // SMBus on P0.0
SBIT(SCL, SFR_P0, 1);           // and P0.1

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void SMBus_Init (void);
void Timer1_Init (void);
void Timer2_Init (void);
void Int0_Init (void);
void Port_Init (void);
INTERRUPT_PROTO(SMBus_ISR, INTERRUPT_SMBUS0);
INTERRUPT_PROTO(ButtonPushed_ISR, INTERRUPT_INT0);
INTERRUPT_PROTO(Timer2_ISR, INTERRUPT_TIMER2);
void RunFreqProg (void);
U8 SetBits (U8 original, U8 reset_mask, U8 new_val);
void ReadStartUpConfiguration (void);
void UpdateStatusOutput (void);
void Update_Flash (U16 addr);
void I2C_ByteWrite (U8 addr, U8 dat);
U8 I2C_ByteRead (U8 addr);
void FLASH_ByteWrite (U16 addr, char byte);
U8 FLASH_ByteRead (U16 addr);
void FLASH_PageErase (U16 addr);


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------
//
// Main routine performs all configuration tasks, then loops forever waiting
// for the 'next' button on the board to be pressed. When pressed, the Si570
// will then output the next frequency.
//
void main (void){
   U8 i;                            // Temporary counter variable used in for loops
   U8 temp_index;                   // Holds index of FOUT1 that was read from flash
   PCA0MD &= ~0x40;                 // WDTE = 0 (disable watchdog timer)
   OSCICN |= 0x03;                  // Configure internal oscillator for
                                    // its maximum frequency (24.5 Mhz)

   // If slave is holding SDA low because of an improper SMBus reset or error
   while(!SDA)
   {
      // Provide clock pulses to allow the slave to advance out
      // of its current state. This will allow it to release SDA.
      XBR1 = 0x40;                  // Enable Crossbar
      SCL = 0;                      // Drive the clock low
      for(i = 0; i < 255; i++);     // Hold the clock low
      SCL = 1;                      // Release the clock
      while(!SCL);                  // Wait for open-drain
                                    // clock output to rise
      for(i = 0; i < 10; i++);      // Hold the clock high
      XBR1 = 0x00;                  // Disable Crossbar
   }
```

```
Port_Init ();                          // Initialize Crossbar and GPIO

CKCON = 0x10;                          // Timer 1 is sysclk
                                       // Timer 2 is sysclk/12 (see TMR2CN)

Timer1_Init ();                        // Configure Timer1 for use as SMBus
                                       // clock source

Timer2_Init ();                        // Configure Timer2 for use with SMBus
                                       // low timeout detect

SMBus_Init ();                         // Configure and enable SMBus

Int0_Init ();                          // Configure INT0

EIE1  |= 0x01;                         // Enable SMBus interrupt
EX0    = 1;                            // Enable External Interrupt 0
EA     = 1;                            // Global interrupt enable  ****MUST BE LAST****

IE0 = 0;                               // Clears INT0 interrupt flag in case previously set

ReadStartUpConfiguration();            // Load initial settings of Si57x/598/599 into the MCU

temp_index = FLASH_ByteRead (FLASH_MEM_ADDR); // Check to see if previous index is stored
                                              // in flash

if (temp_index<FREQ_LIST_LENGTH)    // Checks to see if FLASH_MEM_ADDR contains 0xFF, which
                                    // is what should be initially written to flash. By
                                   // checking for anything greater than FREQ_LIST_LENGTH,
                                    // we can eliminate possibly of invalid value in the
                                    // statically assigned memory location
{
   if (temp_index == 0)
  {
     CURR_FREQ = FREQ_LIST_LENGTH - 1;
  } else {
     CURR_FREQ = temp_index-1;      // Set current frequency to previously stored
                                    // value in flash
   }
   RunFreqProg();                   // Output correct frequency
   UpdateStatusOutput();           // Update LED's based on which freqency in array FOUT1 is
                                    // being output
   if (temp_index==37)              // Set next output frequency, but check to make sure
  {                                 // an overflow won't occur
      CURR_FREQ = 0;
  } else {
     CURR_FREQ = CURR_FREQ + 1;
  }
}
else
{
   CURR_FREQ = FREQ_LIST_LENGTH - 1;// Start up frequency is the last element in the array.
                                    // Set CURR_FREQ index to point to last element
   UpdateStatusOutput();           // Update LED's based on which freqency in array FOUT1 is
                                    // being output
   CURR_FREQ = 0;
}



while(1)                               // Loop forever
{
```

```
      while (NEXT == 0) {}             // Wait for button to be pressed
      RunFreqProg();
     NEXT = 0;                         // Clear NEXT variable
     UpdateStatusOutput();             // Update LEDs on the board
     CURR_FREQ = CURR_FREQ + 1;        // Update counter for next frequency
      Update_Flash (FLASH_MEM_ADDR);   // Update flash with current frequency

     if (CURR_FREQ > (FREQ_LIST_LENGTH - 1))
     {
        CURR_FREQ = 0;                    // Wrap around counter after overflow
         Update_Flash (FLASH_MEM_ADDR);   // Update flash with current frequency
      }
   }
}


//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SMBus_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// The SMBus peripheral is configured as follows:
// - SMBus enabled
// - Slave mode disabled
// - Timer1 used as clock source. The maximum SCL frequency will be
//   approximately 1/3 the Timer1 overflow rate
// - Setup and hold time extensions enabled
// - Free and SCL low timeout detection enabled
//
void SMBus_Init (void)
{
   SMB0CF = 0x5D;                    // Use Timer1 overflows as SMBus clock
                                     // source;
                                     // Disable slave mode;
                                     // Enable setup & hold time extensions;
                                     // Enable SMBus Free timeout detect;
                                     // Enable SCL low timeout detect;

   SMB0CF |= 0x80;                   // Enable SMBus;
}


//-----------------------------------------------------------------------------
// Timer1_Init
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Timer1 is configured as the SMBus clock source as follows:
// - Timer1 in 8-bit auto-reload mode
// - SYSCLK / 12 as Timer1 clock source
// - Timer1 overflow rate => 3 * SMB_FREQUENCY
// - The maximum SCL clock rate will be ~1/3 the Timer1 overflow rate
// - Timer1 enabled
//
void Timer1_Init (void)
{
```

```
// Make sure the Timer can produce the appropriate frequency in 8-bit mode
// Supported SMBus Frequencies range from 10kHz to 100kHz.  The CKCON register
// settings may need to change for frequencies outside this range.
   TMOD = 0x20;                                // Timer1 in 8-bit auto-reload mode
   TH1 = 0xFF - (SYSCLK/SMB_FREQUENCY/3) + 1; // 100kHz or 400kHz for SCL
   TL1 = TH1;                                  // Init Timer1
   TR1 = 1;                                    // Timer1 enabled
}


//-------------------------------------------------------------------------
// Timer2_Init
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Timer2 configured for Si57x/598/599 reset delay as
// follows:
// - Timer2 in 16-bit auto-reload mode
// - SYSCLK/12 as Timer2 clock source
// - Timer2 reload registers loaded for RESET_DELAY_TIME overflow
// - Timer2 pre-loaded to overflow after RESET_DELAY_TIME
//
void Timer2_Init (void)
{
   TMR2CN = 0x00;                      // Timer2 configured for 16-bit auto-
                                       // reload, low-byte interrupt disabled
                                       // Timer2 uses SYSCLK/12 (see CKCON)

   TMR2RLL = 0x5F;
   TMR2RLH = 0x88;
   TMR2L = TMR2RLL;
   TMR2H = TMR2RLH;
   TF2LEN = 0;
   TF2H = 0;
   TF2L = 0;
   ET2  = 1;                           // Timer2 interrupt enable
   TR2 = 1;                            // Start Timer2
}


//-------------------------------------------------------------------------
// Int0_Init
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Configure INT0 to trigger on P0.7
//
void Int0_Init (void)
{
   IT01CF |= 0x07;
   PX0  = 1;                     // Set External Interrupt 0 to high priority
   IT0  = 1;                     // Configure External Interrupt to be edge triggered
}


//-------------------------------------------------------------------------
// Port_Init
//-------------------------------------------------------------------------
//
// Return Value : None
```

```
// Parameters   : None
//
// Configure the Crossbar and GPIO ports.
//
// P0.0   digital   open-drain    SMBus SDA
// P0.1   digital   open-drain    SMBus SCL
// P0.2   digital   open-drain    LED on board to show frequency index
// P0.3   digital   open-drain    LED on board to show frequency index
// P0.4   digital   open-drain    LED on board to show frequency index
// P0.5   digital   open-drain    LED on board to show frequency index
// P0.6   digital   open-drain    LED on board to show frequency index
// P0.7   digital   open-drain    Used as input switch 'next' on board
//
// all other port pins unused
//
// Note: If the SMBus is moved, the SCL and SDA sbit declarations must also
// be adjusted.
//
void Port_Init (void)
{
   P0MDOUT = 0x00;                     // All P0 pins open-drain output

   XBR1 = 0x04;                        // Enable SMBus
   XBR2 = 0x40;                        // Enable crossbar and weak pull-ups

}


//-----------------------------------------------------------------------------
// SMBus Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// SMBus ISR state machine
// - Master only implementation - no slave or arbitration states defined
// - All incoming data is written starting at the global pointer <pSMB_DATA_IN>
// - All outgoing data is read from the global pointer <pSMB_DATA_OUT>
//
INTERRUPT(SMBus_ISR, INTERRUPT_SMBUS0)
{
   bit FAIL = 0;                       // Used by the ISR to flag failed
                                       // transfers

   static char i;                      // Used by the ISR to count the
                                       // number of data bytes sent or
                                       // received

   static bit SEND_START = 0;          // Send a start

   switch (SMB0CN & 0xF0)              // Status vector
   {
      // Master Transmitter/Receiver: START condition transmitted.
      case SMB_MTSTA:
         SMB0DAT = TARGET;             // Load address of the target slave
         SMB0DAT &= 0xFE;              // Clear the LSB of the address for the
                                       // R/W bit
         SMB0DAT |= SMB_RW;            // Load R/W bit
         STA = 0;                      // Manually clear START bit
         i = 0;                        // Reset data byte counter
         break;

      // Master Transmitter: Data byte (or Slave Address) transmitted
      case SMB_MTDB:
         if (ACK)                      // Slave Address or Data Byte
         {                             // Acknowledged?
```

```
         if (SEND_START)
         {
            STA = 1;
            SEND_START = 0;
            break;
         }
         if(SMB_SENDWORDADDR)       // Are we sending the word address?
         {
            SMB_SENDWORDADDR = 0;   // Clear flag
            SMB0DAT = WORD_ADDR;    // Send word address

            if (SMB_RANDOMREAD)
            {
               SEND_START = 1;      // Send a START after the next ACK cycle
               SMB_RW = READ;
            }

            break;
         }

         if (SMB_RW==WRITE)         // Is this transfer a WRITE?
         {

            if (i < SMB_DATA_LEN)   // Is there data to send?
            {
               // send data byte
               SMB0DAT = *pSMB_DATA_OUT;

               // increment data out pointer
               pSMB_DATA_OUT++;

               // increment number of bytes sent
               i++;
            }
            else
            {
              STO = 1;              // Set STO to terminte transfer
              SMB_BUSY = 0;         // Clear software busy flag
            }
         }
         else {}                   // If this transfer is a READ,
                                   // then take no action. Slave
                                   // address was transmitted. A
                                   // separate 'case' is defined
                                   // for data byte recieved.
      }
      else                         // If slave NACK,
      {
         if(SMB_ACKPOLL)
         {
            STA = 1;               // Restart transfer
         }
         else
         {
            FAIL = 1;              // Indicate failed transfer
         }                         // and handle at end of ISR
      }
      break;

   // Master Receiver: byte received
   case SMB_MRDB:
      if ( i < SMB_DATA_LEN )       // Is there any data remaining?
      {
         *pSMB_DATA_IN = SMB0DAT;   // Store received byte
```

```
            pSMB_DATA_IN++;            // Increment data in pointer
            i++;                       // Increment number of bytes received
            ACK = 1;                   // Set ACK bit (may be cleared later
                                       // in the code)

         }

         if (i == SMB_DATA_LEN)        // This is the last byte
         {
            SMB_BUSY = 0;              // Free SMBus interface
            ACK = 0;                   // Send NACK to indicate last byte
                                       // of this transfer
            STO = 1;                   // Send STOP to terminate transfer
         }

         break;

      default:
         FAIL = 1;                     // Indicate failed transfer
                                       // and handle at end of ISR
         break;
   }

   if (FAIL)                           // If the transfer failed,
   {
      SMB0CF &= ~0x80;                 // Reset communication
      SMB0CF |= 0x80;
      STA = 0;
      STO = 0;
      ACK = 0;

      SMB_BUSY = 0;                    // Free SMBus

      FAIL = 0;
   }

   SI = 0;                             // Clear interrupt flag
}


//-----------------------------------------------------------------------------
// External Interrupt 0 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// An external interrupt 0 indicates that the 'next' button on the board
// has been pressed. The ISR sets, 'next' to 1 to allow new frequency to be set
//
INTERRUPT (ButtonPushed_ISR, INTERRUPT_INT0)
{
  NEXT = 1;
}


//-----------------------------------------------------------------------------
// Timer2 Interrupt Service Routine (ISR)
//-----------------------------------------------------------------------------
//
// A Timer2 interrupt indicates an SMBus SCL low timeout.
// The SMBus is disabled and re-enabled if a timeout occurs.
//
INTERRUPT(Timer2_ISR, INTERRUPT_TIMER2)
{
   SMB0CF &= ~0x80;                    // Disable SMBus
   SMB0CF |= 0x80;                     // Re-enable SMBus
```

```
   TF2H = 0;                           // Clear Timer2 interrupt-pending flag
   SMB_BUSY = 0;                       // Free bus
}


//-----------------------------------------------------------------------------
// Support Functions
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// RunFreqProg
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Program Si570 to output the next frequency in the list FOUT1
//
void RunFreqProg (void)
{
   U8 i;                           // Temporary counter variable used in for loops

   U8 n1;                          // Output divider that is modified and used in
                                   // calculating the new RFREQ

   U8 hsdiv;                       // Output divider that is modified and used in
                                   // calculating the new RFREQ

   bit validCombo;                 // Flag that is set to 1 if a valid combination
                                   // of N1 and HS_DIV is found

   U8 reg137;                      // Stores the contents of Register 137
                                   // on Si57x/598/599

   U16 divider_max;                // Maximum divider for HS_DIV and N1 combination

   U16 curr_div;                   // Minimum divider for HS_DIV and N1 combination

//  U16 whole;                      // Used in Double Floating Point Precision Method

   U32 final_rfreq_long;           // Final REFREQ that is sent to the Si57x/598/599

   float curr_n1;                  // Used to calculate the final N1 to send to the
   float n1_tmp;                   // Si570

   float ratio = 0;                // Will hold the final ratio to multiply the initial
                                   // REFREQ by to acheive the final RFREQ

   // Find dividers (get the max and min divider range for the HS_DIV and N1 combo)
   divider_max = FLOORF(FDCO_MAX / FOUT1[CURR_FREQ]);
   curr_div = CEILF(FDCO_MIN / FOUT1[CURR_FREQ]);
   validCombo = 0;

   while (curr_div <= divider_max)
   {
      //check all the HS_DIV values with the next curr_div
      for(i=0; i<6; i++)
      {
         // get the next possible n1 value
         hsdiv = HS_DIV[i];
         curr_n1 = (float)(curr_div) / (float)(hsdiv);

         // Determine if curr_n1 is an integer and an even number or one
         // then it will be a valid divider option for the new frequency
```

```
      n1_tmp = FLOORF(curr_n1);
      n1_tmp = curr_n1 - n1_tmp;
      if(n1_tmp == 0.0)                    // Then curr_n1 is an integer
      {
         n1 = (unsigned char) curr_n1;

         if( (n1 == 1) || ((n1 & 1) == 0) ) // Then the calculated N1 is
         {                                  // either 1 or an even number
            validCombo = 1;
         }
      }
      if(validCombo == 1) break;            // Divider was found, exit loop
   }
   if(validCombo == 1) break;               // Divider was found, exit loop

   curr_div = curr_div + 1;                 // If a valid divider is not found,
                                            // increment curr_div and loop
}

// If validCombo == 0 at this point, then there is an error
// in the calculation. Check if the provided FOUT0 and FOUT1
// are valid frequencies

 // New RFREQ calculation
RFREQ = (FOUT1[CURR_FREQ] * n1 * hsdiv) / FXTAL;

// Calculate RFREQ organizing the float variables to save precision;
// RFREQ is kept as an unsigned long
// only 32 bits are available in the long format
// RFREQ in the device has 34 bits of precision
// only 34 of the 38 bits are needed since RFREQ is between 42.0 and
// 50.0 for fxtal of 114.285MHz (nominal)
ratio = FOUT1[CURR_FREQ] / FOUT0;               // Try to keep ration near 1
                                                // to maintain precision
ratio = ratio * (((float)n1)/((float)INITIAL_N1));
ratio = ratio * (((float)hsdiv)/((float)INITIAL_HSDIV));
final_rfreq_long = ratio * INITIAL_RFREQ_LONG; // Calculate final RFREQ value
                                               // using ratio computed above

for(i = 0; i < 6; i++)
{
   REG[i] = 0;             //clear registers
}

hsdiv = hsdiv - 4;         // Subtract 4 because of the offset of HS_DIV.
                           // Ex: "000" maps to 4, "001" maps to 5

//set the top 3 bits of REG[0] which will correspond to Register 7 on Si57x/598/599
REG[0] = (hsdiv << 5);


// convert new N1 to the binary representation
if(n1 == 1)
{
   n1 = 0;  //Corner case for N1. If N1=1, it is represented as "00000000"
}
else if((n1 & 1) == 0)
{
   n1 = n1 - 1; // If n1 is even, round down to closest odd number. See the
 }              // Si57x/598/599 data sheet for more information.

// Write correct new values to REG[0] through REG[6]
// These will be sent to the Si57x/598/599 and will update the output frequency
REG[0] = SetBits(REG[0], 0xE0, (n1 >> 2));         // Set N1 part of REG[0]
```

```
   REG[1] = (n1 & 3) << 6;                          // Set N1 part of REG[1]
   //Write new version of RFREQ to corresponding registers
   REG[1] = REG[1] | (final_rfreq_long >> 30);
   REG[2] = final_rfreq_long >> 22;
   REG[3] = final_rfreq_long >> 14;
   REG[4] = final_rfreq_long >> 6;
   REG[5] = final_rfreq_long << 2;

/*
   // Double Floating Point Precision Method
   // convert new RFREQ to the binary representation
   // separate the integer part
   whole = FLOORF(RFREQ);

   // get the binary representation of the fractional part
   FRAC_BITS = FLOORF((RFREQ - whole) * POW_2_28);

   // set reg 12 to 10 making frac_bits smaller by
   // shifting off the last 8 bits everytime
   for(i=5; i >=3; i--)
   {
      REG[i] = FRAC_BITS & 0xFF;
      FRAC_BITS = FRAC_BITS >> 8;
   }
   // set the last 4 bits of the fractional portion in reg 9
   REG[2] = SetBits(REG[2], 0xF0, (FRAC_BITS & 0xF));

   // set the integer portion of RFREQ across reg 8 and 9
   REG[2] = SetBits(REG[2], 0x0F, (whole & 0xF) << 4);
   REG[1] = SetBits(REG[1], 0xC0, (whole >> 4) & 0x3F);
*/

   reg137 = I2C_ByteRead(137);              // Read the current state of Register 137

   I2C_ByteWrite(137, reg137 | 0x10);       // Set the Freeze DCO bit in that register
                                            // This must be done in order to update
                                            // Registers 7-12 on the Si57x/598/599
   for(i=0; i<6; i++)
   {
      I2C_ByteWrite(i+7, REG[i]);           // Write the new values to Registers 7-12
   }

   reg137 = I2C_ByteRead(137);              // Read the current state of Register 137

   I2C_ByteWrite(137, reg137 & 0xEF);       // Clear the Freeze DCO bit

   I2C_ByteWrite(135, 0x40);                // Set the NewFreq bit to alert the DPSLL
                                            // that a new frequency configuration
                                            // has been applied
}


//---------------------------------------------------------------------------
// SetBits
//---------------------------------------------------------------------------
//
// Return Value : unsigned char
// Parameters   :
//   1) unsigned char original - original state of variable to be changed
//
//   2) unsigned char reset_mask - contains mask of bits that will reset the
//                  the original char variable.
//
//   3) unsigned char new_val - contains a mask of bits that need to be set in
```

```
//                      the original variable.
//
// This function sets appropriate bits in an unsigned char variable
//
U8 SetBits(U8 original, U8 reset_mask, U8 new_val)
{
   return (( original & reset_mask ) | new_val );
}


//-----------------------------------------------------------------------------
// ReadStartUpConfig
//-----------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// Reads start-up register contents for RFREQ, HS_DIV, and N1 and calculates
// the internal crystal frequency (FXTAL)
//
void ReadStartUpConfiguration (void)
{
    U8 i;                           // Counter used in for loops
   I2C_ByteWrite(135, 0x01);        // Writes 0x01 to register 135. This
                                    // will recall NVM bits into RAM. See
                                    // register 135 in Si57x/598/599 data sheet for
                                    // more information

   // read registers 7 to 12 of Si57x/598/599
   // REG[0] is equivalent to register 7 in the device
   // REG[5] is register 12 in the device
   for(i=0; i<6; i++)
   {
      REG[i] = I2C_ByteRead(i+7);
   }

   INITIAL_HSDIV = ((REG[0] & 0xE0) >> 5) + 4; // Get value fo INITIAL_HSDIV from REG[0]
   // 4 is added to this because the bits "000" correspond to an HSDIV of 4, so there is
   // an offset of 4. See the register 7 of Si570 datasheet for more information.

   INITIAL_N1 = (( REG[0] & 0x1F ) << 2 ) + (( REG[1] & 0xC0 ) >> 6 );
   // Get correct value of INITIAL_N1 by adding parts of REG[0] and REG[1]

   if(INITIAL_N1 == 0)
   {
      INITIAL_N1 = 1;               // This is a corner case of N1
   }
   else if(INITIAL_N1 & 1 != 0)
   {
      INITIAL_N1 = INITIAL_N1 + 1;  // As per datasheet, illegal odd divider values should
                                    // should be rounded up to the nearest even value.
   }

   // Double Floating Point Precision Method
   // RFREQ conversion (reconstruct the fractional portion (bits 0 to 28) from the registers)
   // (this method requires double precision floating point data type to be accurate)
/* FRAC_BITS = (( REG[2] & 0xF ) * POW_2_24 );
   FRAC_BITS = FRAC_BITS + (REG[3] * POW_2_16);
   FRAC_BITS = FRAC_BITS + (REG[4] * 256);
   FRAC_BITS = FRAC_BITS + REG[5];

   RFREQ = FRAC_BITS;
    RFREQ = RFREQ / POW_2_28;
*/
```

```
   // Read initial value for RFREQ. A 34-bit number is fit into a 32-bit space
   // by ignoring lower 2 bits.
   INITIAL_RFREQ_LONG = ( REG[1] & 0x3F );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[2] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[3] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 8) + ( REG[4] );
   INITIAL_RFREQ_LONG = (INITIAL_RFREQ_LONG << 6) + ( REG[5] >> 2 );

   // RFREQ conversion (reconstruct the integer portion from the registers)
   RFREQ = RFREQ + ( (( REG[1] & 0x3F ) << 4 ) + (( REG[2] & 0xF0 ) >> 4 ) );

   // Crystal Frequency (FXTAL) calculation
   FXTAL = (FOUT0 * INITIAL_N1 * INITIAL_HSDIV) / RFREQ;          //MHz
}


//-------------------------------------------------------------------------
// UpdateStatusOutput
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : None
//
// This function updates status of the LED's on the board by reading the
// the variable currFreq and updating P0. The LED's use a binary counting
// system. Writing a 0 to the corresponding pin in Port 0 will light up LED
//
void UpdateStatusOutput (void)
{
   P0 = (0x80 | ((~CURR_FREQ) << 2));
}


//-------------------------------------------------------------------------
// Update_Flash
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   : Target flash memory address
//
// This function writes the value of CURR_FREQ to the memory location in
// flash that is specified by the global definition of FLASH_MEM_ADDR
//
// In flash, you can only clear bits, which is why a flash page erase is
// needed before each new write to the desired flash memory location.
//
void Update_Flash (U16 addr)
{
   FLASH_PageErase (FLASH_MEM_ADDR);
   FLASH_ByteWrite (FLASH_MEM_ADDR, CURR_FREQ);
}


//-------------------------------------------------------------------------
// I2C_ByteWrite
//-------------------------------------------------------------------------
//
// Return Value : None
// Parameters   :
//   1) unsigned char addr - address to write in the device via I2C
//      range is full range of character: 0 to 255
//
```

```
//   2) unsigned char dat - data to write to the address <addr> in the device
//      range is full range of character: 0 to 255
//
// This function writes the value in <dat> to location <addr> in the device
// then polls the device until the write is complete.
//
void I2C_ByteWrite (U8 addr, U8 dat)
{
   while (SMB_BUSY);                   // Wait for SMBus to be free.
   SMB_BUSY = 1;                       // Claim SMBus (set to busy)

   // Set SMBus ISR parameters
   TARGET = SLAVE_ADDR;                // Set target slave address
   SMB_RW = WRITE;                     // Mark next transfer as a write
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave Address
   SMB_RANDOMREAD = 0;                 // Do not send a START signal after
                                       // the word address
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling (The ISR
                                       // will automatically restart the
                                       // transfer if the slave does not
                                       // acknoledge its address.

   // Specify the Outgoing Data
   WORD_ADDR = addr;                   // Set the target address in the
                                       // device's internal memory space

   SMB_SINGLEBYTE_OUT = dat;           // Store <dat> (local variable) in a
                                       // global variable so the ISR can read
                                       // it after this function exits

   // The outgoing data pointer points to the <dat> variable
   pSMB_DATA_OUT = &SMB_SINGLEBYTE_OUT;

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte

   // Initiate SMBus Transfer
   STA = 1;
}


//-----------------------------------------------------------------------------
// I2C_ByteRead
//-----------------------------------------------------------------------------
//
// Return Value :
//   1) unsigned char data - data read from address <addr> in the device
//      range is full range of character: 0 to 255
//
// Parameters   :
//   1) unsigned char addr - address to read data from the device
//      range is full range of character: 0 to 255
//
// This function returns a single byte from location <addr> in the device then
// polls the <SMB_BUSY> flag until the read is complete.
//
U8 I2C_ByteRead (U8 addr)
{
   U8 return_val;                      // Holds the return value

   while (SMB_BUSY);                   // Wait for SMBus to be free.
   SMB_BUSY = 1;                       // Claim SMBus (set to busy)

   // Set SMBus ISR parameters
```

```
   TARGET = SLAVE_ADDR;                // Set target slave address
   SMB_RW = WRITE;                     // A random read starts as a write
                                       // then changes to a read after
                                       // the repeated start is sent. The
                                       // ISR handles this switchover if
                                       // the <SMB_RANDOMREAD> bit is set.
   SMB_SENDWORDADDR = 1;               // Send Word Address after Slave Address
   SMB_RANDOMREAD = 1;                 // Send a START after the word address
   SMB_ACKPOLL = 1;                    // Enable Acknowledge Polling


   // Specify the Incoming Data
   WORD_ADDR = addr;                   // Set the target address in the
                                       // devices's internal memory space

   pSMB_DATA_IN = &return_val;         // The incoming data pointer points to
                                       // the <retval> variable.

   SMB_DATA_LEN = 1;                   // Specify to ISR that the next transfer
                                       // will contain one data byte

   // Initiate SMBus Transfer
   STA = 1;
   while(SMB_BUSY);                    // Wait until data is read

   return return_val;

}


//-----------------------------------------------------------------------------
// FLASH Routines
//-----------------------------------------------------------------------------
//
// The following functions were taken from Application Note 201 Software for
// 'F30x devices in the file F300_FlashPrimitices.c. It can be found on the
// Silicon Labs website at www.silabs.com
//
//-----------------------------------------------------------------------------
// FLASH_ByteWrite
//-----------------------------------------------------------------------------
//
// This routine writes <byte> to the linear FLASH address <addr>.
//
void FLASH_ByteWrite (U16 addr, char byte)
{
   bit EA_SAVE = EA;                   // preserve EA

   // FLASH write pointer
   SEGMENT_VARIABLE_SEGMENT_POINTER(pwrite, char, SEG_XDATA, SEG_DATA);

   EA = 0;                             // disable interrupts

   RSTSRC = 0x06;                      // enable VDD monitor as a reset source

   pwrite = (char SEG_XDATA *) addr;

   FLKEY  = 0xA5;                      // Key Sequence 1
   FLKEY  = 0xF1;                      // Key Sequence 2
   PSCTL |= 0x01;                      // PSWE = 1

   RSTSRC = 0x06;                      // enable VDD monitor as a reset source

   *pwrite = byte;                     // write the byte
```

```
   PSCTL &= ~0x01;                     // PSWE = 0

   EA = EA_SAVE;                       // restore interrupts
}


//-----------------------------------------------------------------------------
// FLASH_ByteRead
//-----------------------------------------------------------------------------
//
// This routine reads a <byte> from the linear FLASH address <addr>.
//
U8 FLASH_ByteRead (U16 addr)
{
   bit EA_SAVE = EA;                   // preserve EA

   // FLASH write pointer
   SEGMENT_VARIABLE_SEGMENT_POINTER(pread, char, SEG_CODE, SEG_DATA);
   U8 byte;

   EA = 0;                             // disable interrupts

   pread = (char SEG_CODE *) addr;

   byte = *pread;                      // read the byte

   EA = EA_SAVE;                       // restore interrupts

   return byte;
}


//-----------------------------------------------------------------------------
// FLASH_PageErase
//-----------------------------------------------------------------------------
//
// This routine erases the FLASH page containing the linear FLASH address
// <addr>.
//
void FLASH_PageErase (U16 addr)
{
   bit EA_SAVE = EA;                    // preserve EA

   // FLASH write pointer
   SEGMENT_VARIABLE_SEGMENT_POINTER(pwrite, char, SEG_XDATA, SEG_DATA);

   EA = 0;                            // disable interrupts


   RSTSRC = 0x06;                     // enable VDD monitor as a reset source

   pwrite = (char SEG_XDATA *) addr;

   FLKEY  = 0xA5;                     // Key Sequence 1
   FLKEY  = 0xF1;                     // Key Sequence 2
   PSCTL |= 0x03;                     // PSWE = 1; PSEE = 1

   RSTSRC = 0x06;                     // enable VDD monitor as a reset source
   *pwrite = 0;                       // initiate page erase

   PSCTL &= ~0x03;                    // PSWE = 0; PSEE = 0

   EA = EA_SAVE;                      // restore interrupts
}
```

DOCUMENT CHANGE LIST

Revision 0.1 to Revision 0.2

- Revised Reference Design Software A to comply with firmware guidelines
- Added Reference Design Software B
- Updated Introduction, Compiler, and Solution sections

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.

400 West Cesar Chavez
Austin, TX 78701
Tel:1+ (512) 416-8500
Fax:1+ (512) 416-9669
Toll Free:1+ (877) 444-3032

Email: VCXOinfo@silabs.com
Internet: www.silabs.com