```
/*
 * File:   LogicalDrv.h
 * Author: Alec
 *
 * Created on September 18, 2021, 7:35 AM
 */

#ifndef LOGICALDRV_H
#defineLOGICALDRV_H

#ifdef__cplusplus
extern "C" {
#endif

#include <stdbool.h>

#include "SPIMemory.h"
#include "Utility.h"

#define LOGICAL_SECTOR_SIZE                         128L

/* Physical Format used in the SPI flash
 *
 *      PHYSICAL_SECTOR_HEADER 0 @ ADDRESS 0 OF DEVICE
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      ,,,
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      unused gap at end of physical sector
 *      PHYSICAL_SECTOR_HEADER 1 ON NEXT HD_PHYSICAL_SECTOR_SIZE BOUNDARY
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      LOGICAL_SECTOR_HEADER + LOGICAL_SECTOR_SIZE bytes of data
 *      ,,,
 */

// Sector related types
typedef UINT16 Sector_T;
typedef UCHAR SectorBuffer_T[LOGICAL_SECTOR_SIZE];



// Physical Sector Header
typedef struct __attribute__((packed, aligned(1))) {
    UINT32      writeCycles;            // total write cycles to this physical sector since last disk format
    Sector_T    sectorReMap;            // if != NO_SECTOR_REMAP then this sector has been re-mapped to the specified sector
        #define     NO_SECTOR_REMAP     0xffff
    Sector_T    linkedList;             // linked list pointer to next sector used for list of free remap physical sectors
        // value of linkedList in the last record of the list
        // IMPORTANT - the code relies on the fact that a flash '1' can be changed to a '0' by writing (but NOT vis-versa)
        //             by using all zeros as the terminator, the previous (non-zero) linked list physical sector value can
        //             be changed to zero WITHOUT having to erase the physical sector.  Do NOT change this value.
        #define     LINKED_LIST_TERMINATOR      0x0000
} PHYSICAL_SECTOR_HEADER, *PPHYSICAL_SECTOR_HEADER;


// Logical Sector Header
typedef struct __attribute__((packed, aligned(1))) {
    UINT16      checkSum;               // sector data checksum/CRC
} LOGICAL_SECTOR_HEADER, *PLOGICAL_SECTOR_HEADER;


// data structure used in the DRIVER_LOGICAL_SECTOR
// this is where the driver should store any important data that it needs
typedef struct __attribute__((packed, aligned(1))) {
    UCHAR       driveFormatted;
        #define     DRIVE_FORMATTED     0x33
    // the rest of LOGICAL_SECTOR_SIZE is currently unused
} DRIVER_LOGICAL_SECTOR_LAYOUT, *PDRIVER_LOGICAL_SECTOR_LAYOUT;


// total #of logical sectors that exist on the drive
#define HD_NUM_LOGICAL_SECTORS                      (((HD_PHYSICAL_SECTOR_SIZE - sizeof(PHYSICAL_SECTOR_HEADER)) / LOGICAL_SECTOR_SIZE) * HD_NUM_PHYSICAL_SECTORS)


// macro to get physical sector address from logical sector address
// physical sector is always at the start of each HD_PHYSICAL_SECTOR_SIZE block
#define LOGICAL_TO_PHYSICAL_ADDRESS(logicalAddress)  (logicalAddress & ((SPI_ADDRESS_TYPE) ~(HD_PHYSICAL_SECTOR_SIZE - 1)))

// macros to convert between physical sector address and sector number
#define PHYSICAL_SECTOR_TO_ADDRESS(physicalSector)   (physicalSector * HD_PHYSICAL_SECTOR_SIZE)
#define PHYSICAL_ADDRESS_TO_SECTOR(physicalAddress)  ((Sector_T) (physicalAddress / HD_PHYSICAL_SECTOR_SIZE))


// macro to determine logical sector relative offset within the physical sector
// (ie: the first logical sector offset = 0, second = 1, etc...)
#define LOGICAL_SECTOR_OFFSET(logical_sector_address)   ((UINT16) (((logical_sector_address - sizeof(PHYSICAL_SECTOR_HEADER)) % HD_PHYSICAL_SECTOR_SIZE) / LOGICAL_SECTOR_FRAME_SIZE))

// Drive status as reported to OS
typedef enum{
    DRIVE_STATUS_NO_WEAR = 0,               // reported from drive format until a physical sector has been remapped
    DRIVE_STATUS_NO_MEDIA,
    DRIVE_STATUS_DEFECTIVE_MEDIA,
    DRIVE_STATUS_UNFORMATTED_MEDIA,
    DRIVE_STATUS_WEAR_LEVEL_LOW,
    DRIVE_STATUS_WEAR_LEVEL_MEDIUM,
    DRIVE_STATUS_WEAR_LEVEL_HIGH
```

```
    DRIVE_STATUS_WEAR_LEVEL_FAILURE,
    DRIVE_STATUS_HARDWARE_FAILURE
} DriveStatus_T;

// Drive parameters as reported to OS (length = 6)
typedef struct __attribute__((packed, aligned(1))) {
    DriveStatus_T   driveStatus;            // drive status
    UINT16          bytesPerSector;         // #of bytes per sector
    Sector_T        sectors;                // #of sectors on the logical drive
    UCHAR           sectorsPerBlock;        // #of sectors per block (minimum #of sectors that can be written at once)
} DriveParameters_T, *PDriveParameters_T;


// CP/M Drive Structures - delete when done
#if 0
typedef struct{
    UINT16  sectorPerTrack;
    UCHAR   blockShift;         // BLKSHFT *SEE BELOW
    UCHAR   blockMask;          // BLKMASK
    UCHAR   extentMask;
    UINT16  diskSize;           // (#of blocks - 1)
    UINT16  dirSize;
    UINT16  alloc0;             // storage for 1st bytes of bit map (dir space used)
    UINT16  alloc1;
    UINT16  trkOffset;          // first usable track number
    UINT16  pSectorXlateTable;  // local (not provided by BIOS?)
} PARAMETER_BLOCK;


/* Block shift is a single bit indicating how many sectors are in a block?
 * (0x01 = 2, 0x02 = 4, 0x04 == 8, 0x08 = 16, ... 0x80 = 256)
 *
 * block number is the physical sector desired.
 *
 * block number = (saveext * (2^(8-blockshift)))
 *                 + (savenrec / (2^blockshift))
 *
 * saveext = disk extent number
 * savenrec = disk record number (sector?)
 */


typedef struct{
    UINT16  pTranslationTable;
    UINT16  pScratch1;
    UINT16  pScratch2;
    UINT16  pScratch3;
    UINT16  pDirBuffer;
    UINT16  pParmBlock;     // ptr to PARAMETER_BLOCK (block is 15 bytes long?)
    UINT16  checkVector;
    UINT16  allocationVector;

} DRIVE_PARAMETER_BLOCK;
#endif

// sector buffer used to read and write logical sectors
extern SectorBuffer_T sectorBuffer;
// drive parameters
extern DriveParameters_T driveParameters;

void LogicalDrv_Init(void);
void Format_Logical_Drive(void);
bool Read_Logical_Sector(Sector_T sector, SectorBuffer_T rBuffer);
bool Write_Logical_Sector(Sector_T sector, SectorBuffer_T wBuffer);
bool Erase_Logical_Sector(Sector_T sector);
const Sector_T OS_Sector_Limit(void);


#ifdef __cplusplus
}
#endif

#endif/* LOGICALDRV_H */
```

```
/*
 * File:   RegisterIF.h
 * Author: Alec
 *
 * Created on July 21, 2021, 3:50 PM
 */

#ifndef REGISTERIF_H
#defineREGISTERIF_H

#ifdef__cplusplus
extern "C" {
#endif

#include <stdbool.h>

void RegisterIF_Init(void);
void RegisterIF_Process(void);
void RegisterIF_Timer(void);


// IF Register Definitions based on board feature type
// Register Definition
//   Read register 0 - read status
//   Read register 1 - read data
//   Read register 2 - undefined
//   Read register 3 - read board feature type
//   Write register 0 - write command
//   Write register 1 - write data
//   Write register 2 - UNDEFINED
//   Write register 3 - UNDEFINED

// Read of Register 3 returns the feature code for all boards as defined below
#define IF_FEATURE_KEYBOARD     0xa0
    // Read Register 0 return Status byte as follows
    #define KYBD_FOUND          0x01    /* set if a functioning keyboard has been detected */
    #define KYBD_HARDERR        0x02    /* set if a hardware error was detected */
    #define KYBD_CAPS_LOCK      0x04    /* set if caps lock is currently active */
    #define KYBD_NUM_LOCK       0x08    /* set if num lock is currently active */
    #define KYBD_SCROLL_LOCK    0x10    /* set if scroll lock is currently active */
    #define KYBD_CONTROL_ACTIVE 0x20    /* set if 'Ctrl' key is currently depressed */
    #define KYBD_ALT_ACTIVE     0x40    /* set if 'Alt' key is currently depressed */
    #define KYBD_SHIFT_ACTIVE   0x80    /* set if 'Shift' key is currently depressed */
    // Read Register 1 returns the next 'key' pressed ASCII value
    #define KYBD_NOKEY_PRESSED  0x00    /* value returned when no key is pressed */
    // Read Register 2 returns - UNDEFINED not yet implemented
#define IF_FEATURE_HARDDISK     0xa1


// Max inter-byte time delay allowed
#define MAX_INTERBYTE_MS    500

// security key to prevent accidental device formats
#define HDIF_FORMAT_KEY     0xaa55
```

```
// valid commands supported
typedef enum{
    HDIF_CMD_READ_SECTOR = 0xa3,
    HDIF_CMD_WRITE_SECTOR,
    HDIF_CMD_FORMAT_DEVICE,
    HDIF_CMD_GET_DRIVE_STATUS
} HDIF_CMDS_SUPPORTED;


/* Host must access the IF as follows.
 *
 * For HDIF_CMD_READ_SECTOR & HDIF_CMD_WRITE_SECTOR
 * Write command structure to write register 0 (command):
 *
 *   CMD, LSB SECTOR, MSB SECTOR, LSB OFFSET, MSB OFFSET, LSB #OF BYTES, MSB #OF BYTES, ... followed by
 *
 *      Write "#OF BYTES" to write register 1 (data)
 *
 * For HDIF_CMD_FORMAT_DEVICE
 *
 * Write command structure to write register 0 (command):
 *
 *   CMD, 0x55, 0xaa    (HDIF_FORMAT_KEY)
 */

#ifdef __cplusplus
}
#endif

#endif/* REGISTERIF_H */
```

```
/*
 * File:   RemapPins.h
 * Author: Alec
 *
 * Created on August 13, 2021, 4:35 PM
 */

#ifndef REMAPPINS_H
#defineREMAPPINS_H

#ifdef__cplusplus
extern "C" {
#endif

// TODO - see if these define values are not already defined somewhere in the compiler
//        if so, update code and get rid of these.

// define ports for input mapping
#define PPS_PORTA   0x00
#define PPS_PORTB   0x01
#define PPS_PORTC   0x02

// define functions for output mapping
#define PPS_LATxy       0x00
#define PPS_CCP1        0x01
#define PPS_CCP2        0x02
#define PPS_PWM3        0x03
#define PPS_PWM4        0x04
#define PPS_TX1CK1      0x05        /* EUSART 'TX1' output | EUSART 'CK1' output */
#define PPS_DT1         0x06        /* EUSART 'DT1' output part of bi-directional pin */
#define PPS_SCL1SCK1    0x07        /* I2C 'SCL' output part of bi-directional pin | SPI 'SCLK' output */
#define PPS_SDA1SDO1    0x08        /* I2C 'SDA' output part of bi-directional pin | SPI 'SDO' output */
#define PPS_TMR0        0x09



#ifdef__cplusplus
}
#endif

#endif/* REMAPPINS_H */
```

```
/*
 * File:   ShiftOut.h
 * Author: Alec
 *
 * Created on August 13, 2021, 4:58 PM
 */

#ifndef SHIFTOUT_H
#defineSHIFTOUT_H

#ifdef__cplusplus
extern "C" {
#endif

// The following assume that all SPI pins are configured on PORTC
// NOTE: not all code uses these, if you move the feature you still have to modify code
//       (this was done to keep code faster and more readable)
#define SHIFT_DATA_OUT_PORT_PIN 0   /* data out is on RC0 */
#define SHIFT_CLK_PORT_PIN      1   /* clock is on RC1 */
#define OUT_LATCH_CLK_PORT_PIN  2   /* output latch clock is on RC2 */
#define SHIFT_DATA_IN_PORT_PIN  3   /* data in is on RC3 */
#define IN_LATCH_MODE_PORT_PIN  4   /* input latch mode (0=load, 1=shift) is on RC4 */

extern unsigned char ShiftOutValues[1];
extern unsigned char ShiftInValues[1];

extern void ShiftOut_Init(void);

// Force the values out to the physical hardware
extern void ShiftOut(void);

#ifdef__cplusplus
}
#endif

#endif/* SHIFTOUT_H */
```

```
/*
 * File:   SPIFlash.h
 * Author: Alec
 *
 * Created on September 14, 2021, 8:35 AM
 *
 */

#ifndef SPIFLASH_H
#defineSPIFLASH_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Utility.h"

// Physical disk parameters
#define HD_PHYSICAL_SECTOR_SIZE         4096L
#define HD_NUM_PHYSICAL_SECTORS         1024L
#define FLASH_MAX_WRITE_CYCLES          100000
#define FLASH_PAGE_SIZE                 256


// SPI RAM
#define SPI_RAM_SIZE                    (64 * 1024L)

typedef unsigned long int SPI_ADDRESS_TYPE;

// Commands supported by the SPI flash chip
typedef enum{
    WRITE_ENABLE = 0x06,
    WRITE_DISABLE = 0x04,
    MFGR_DEVICE_ID = 0x90,
    READ_DATA = 0x03,
    PAGE_PROGRAM = 0x02,
    SECTOR_ERASE_4K = 0x20,
    BLOCK_ERASE_32K = 0x52,
    BLOCK_ERASE_64K = 0xd8,
    CHIP_ERASE1 = 0xc7,
    CHIP_ERASE2 = 0x60,

    READ_STATUS_REG1 = 0x05,
```

```
    WRITE_STATUS_REG1 = 0x01,
    READ_STATUS_REG2 = 0x35,
    WRITE_STATUS_REG2 = 0x31,
    READ_STATUS_REG3 = 0x15,
    WRITE_STATUS_REG3 = 0x11,

    /* more commands exist, add them later if needed */

    DUMMY_COMMAND = 0x00

} SPIF_Command;

typedef enum{
    SRP = 0x80,         /* Status Register Protect */
    SEC = 0x40,         /* SECtor protect */
    TB = 0x20,          /* Top/Bottom protect */
    BP2 = 0x10,         /* Block Protect bit 2 */
    BP1 = 0x08,         /* Block Protect bit 1 */
    BP0 = 0x04,         /* Block Protect bit 0 */
    WEL = 0x02,         /* Write Enable Latch */
    BUSY = 0x01         /* BUSY - write in progress flag */
} SPIF_Status_Reg1_Bits;

// SPI RAM reg1 (mode) values
#define SPIR_BYTE_MODE          0x00
#define SPIR_PAGE_MODE          0x80
#define SPIR_SEQUENTIAL_MODE    0x40        /* default */

typedef enum{
    SUS = 0x80,         /* SUSpend status */
    CMP = 0x40,         /* CoMPlement protect */
    LB3 = 0x20,         /* Security Register Lock Bit 3 */
    LB2 = 0x10,         /* Security Register Lock Bit 2 */
    LB1 = 0x08,         /* Security Register Lock Bit 1 */
    QE = 0x02,          /* Quad Enable */
    SRL = 0x01          /* Status Register Lock */
} SPIF_Status_Reg2_Bits;

typedef enum{
    DRV1 = 0x40,        /* Output Driver Strength bits (see below) */
    DRV0 = 0x20,        /* 00=100%, 01=75%, 10=50%, 11=25% */
    WPS = 0x04          /* Write Protect Selection */
} SPIF_Status_Reg3_Bits;
```

```
void SPIMemory_Init(void);

// configuration functions
void Map_SPI_To_Flash(void);
void Map_SPI_To_RAM(void);
void Map_SPI_To_IO(void);

// common device functions to both Flash and RAM
UINT16 Get_SPI_Device_ID(void);
unsigned char Read_SPI_Register(UCHAR which);
void Write_SPI_Register(UCHAR which, UCHAR value);
void Read_SPI_Data(SPI_ADDRESS_TYPE from, UINT16 length, unsigned char *buffer);
void Write_SPI_Data(SPI_ADDRESS_TYPE from, UINT16 length, unsigned char *buffer);

// Flash specific functions
void SPI_Write_Enable(bool enable);
bool SPI_Device_Busy(void);
bool SPI_Write_Enabled(void);
void Erase_SPI_Sector(SPI_ADDRESS_TYPE address);
void Erase_SPI_Chip(void);
bool SPI_RAM_Test(void);



#ifdef __cplusplus
}
#endif

#endif/* SPIFLASH_H */
```

```
/*
 * File:   Timer.h
 * Author: Alec
 *
 * Created on June 8, 2021, 6:41 AM
 */

#ifndef TIMER_H
#defineTIMER_H

#ifdef__cplusplus
extern "C" {
#endif

// Timer Initialization
void Timer_Init(void);

// Timer interrupt handler
void Timer0_Handler(void);

// get Timer0 uS period (define preferred - as function generates code even though it is 'const')
#define TIMER0USPERIOD      65280
inline const unsigned short int Timer0uSPeriod(void);

#if TIMER0USPERIOD < 1000
#error TIMER0USPERIOD must be at least 1mS
#endif

// Define #of ticks needed to guarantee at least 'mS' milli-seconds expire before the countdown completes
#define mSToTicks(mS)   (((mS) / (TIMER0USPERIOD / 1000)) + 1)

#ifdef__cplusplus
}
#endif

#endif/* TIMER_H */
```

```
/*
 * File:   Utility.h
 * Author: Alec
 *
 * Created on August 13, 2021, 4:36 PM
 */

#ifndef UTILITY_H
#defineUTILITY_H

#ifdef__cplusplus
extern "C" {
#endif

#define NUM_ENTRIES(x)  (sizeof(x)/sizeof(x[0]))

typedef unsigned char UCHAR;
typedef unsigned short int UINT16;
typedef unsigned long int UINT32;

#ifdef__cplusplus
}
#endif

#endif/* UTILITY_H */
```

```
/*
 * File:   main.c
 * Author: Alec
 *
 * Created on June 3, 2021, 6:41 AM
 */

// CONFIG
#pragma config FEXTOSC =    OFF
#pragma config RSTOSC =     HFINTOSC_32MHZ
#pragma config CLKOUTEN =   ON
#pragma config VDDAR =      HI
#pragma config MCLRE =      EXTMCLR
#pragma config PWRTS =      PWRT_64
#pragma config WDTE =       OFF
#pragma config BOREN =      OFF
#pragma config BORV =       LO
#pragma config PPS1WAY =    ON
#pragma config STVREN =     ON

#pragma config BBSIZE =     BB512
#pragma config BBEN =       OFF
#pragma config SAFEN =      OFF
#pragma config WRTAPP =     OFF
#pragma config WRTB =       OFF
#pragma config WRTC =       OFF
#pragma config WRTSAF =     OFF
#pragma config LVP =        ON

#pragma config CP =         OFF


#include <xc.h>
```

```
#include <stdint.h>

#include "RegisterIF.h"
#include "ShiftOut.h"
#include "SPIMemory.h"
#include "Timer.h"
#include "LogicalDrv.h"



#define SUPPORT_RUDIMENTARY_FLASH_TEST    0

// main interrupt handler
void __interrupt() Hardware_Interrupt_Vector(void){

    // invoke other interrupt modules
    Timer0_Handler();
}

#if SUPPORT_RUDIMENTARY_FLASH_TEST
unsigned char Reg[3];

#define TESTDATALEN 10
const char TestData[TESTDATALEN] = "Test Data ";
#endif

// main program entry point
void main(void) {

#if SUPPORT_RUDIMENTARY_FLASH_TEST
    unsigned char buffer[TESTDATALEN];
#endif

    // Set all bits to digital (not analog) functions)
```

```
    ANSELA = 0x00;
    ANSELB = 0x00;
    ANSELC = 0x00;

    // Initialize all the modules
    ShiftOut_Init();
    RegisterIF_Init();
    SPIMemory_Init();
    Timer_Init();
    LogicalDrv_Init();

    ei(); // enable all interrupts

#if SUPPORT_RUDIMENTARY_FLASH_TEST
    Map_SPI_To_Flash();


    for (int x=0; x<TESTDATALEN; x++) buffer[x] = 0x00;

    Write_SPI_Data(0, TESTDATALEN, (unsigned char *) TestData);
    Read_SPI_Data(0, TESTDATALEN, buffer);

    Erase_SPI_Sector(0);

    Read_SPI_Data(0, TESTDATALEN, buffer);
#endif

    while (1){

#if SUPPORT_RUDIMENTARY_FLASH_TEST
        unsigned short int flashID;

        Reg[0] = Read_SPI_Register(1);
        Reg[1] = Read_SPI_Register(2);
```

```
        Reg[2] = Read_SPI_Register(3);



        flashID = Get_SPI_Device_ID();
#endif

        // Handle any Host requests
        RegisterIF_Process();
    }
}
```

```
/*
 * File:   RegisterIF.h
 * Author: Alec
 *
 * Created on July 21, 2021, 3:50 PM
 */

#include <pic16f15244.h>
#include "RegisterIF.h"
#include "RemapPins.h"
#include "ShiftOut.h"
#include "SPIMemory.h"
#include "Timer.h"
#include "Utility.h"
#include "LogicalDrv.h"

// max #of timer ticks allowed between host bytes in a command
#define MAX_INTERBYTE_TICKS     mSToTicks(MAX_INTERBYTE_MS)

// read data value returned if more data read than requested, OR if read aborted due to a timeout
// making this an uncommon (not 0) number aids in trouble-shooting host access issues.
#define DUMMY_READ_DATA 0xa5

typedef enum{
    CMD_IDLE = 0,           /* waiting for command */

    // READ/WRITE command states
    CMD_RW_LSB_SECTOR,
    CMD_RW_MSB_SECTOR,
    CMD_RW_LSB_NUMBYTES,
    CMD_RW_MSB_NUMBYTES,
    CMD_RW_PROCESSING_DATA
} HDIF_CMD_STATE;


//****************************
// Feature Specific Variables
//****************************

void Abort_Command(void);
void Process_WriteCmd(void);
void Process_WriteData(void);
void Process_ReadData(void);
void Start_Timeout();
void Stop_Timeout();

void RegisterIF_Handle_Request(void);

unsigned char Last_ShiftOut_Value = 0xff;

unsigned int IF_Timer;
HDIF_CMD_STATE cmdState;

UCHAR cmdReceived;
Sector_T sector;
UINT16 dataBytesRequested;
UINT16 dataBytesProcessed;
```

```
//****************************
// Always Needed Variables
//****************************

// INPUTS
// ======
// RA2 - Chip select input
// RC5 - A0 input
// RC6 - A1 input
// RC7 - !IORD

// OUTPUTS
// =======
// RA4 - system clock output
// RA5 - clock enable

#define INPUT_TYPE_MASK     0xE0
    #define INPUT_RD_ADDR0      0x00
    #define INPUT_RD_ADDR1      0x20
    #define INPUT_RD_ADDR2      0x40
    #define INPUT_RD_ADDR3      0x60
    #define INPUT_WR_ADDR0      0x80
    #define INPUT_WR_ADDR1      0xA0
    #define INPUT_WR_ADDR2      0xC0
    #define INPUT_WR_ADDR3      0xE0

// local copies of the last values the Host wrote
unsigned char WriteRegs[4];

// Initialization
void RegisterIF_Init(void){

//************************
// Always Needed Logic
//************************

    // system clock / 4 is routed to pin RA4 by way of the configuration bits as follows in main.c
    //      #pragma config CLKOUTEN =   ON

    // IMPORTANT - the CLKEN line can not be enabled in the interrupt.  The reason for this
    // is that the PIC interrupt latency is too long and by the time CLKEN was enabled, the
    // host instruction cycle would have terminated without our being able to insert wait states.
    //
    // The proper way to handle CLKEN is to have it enabled WHILE we are waiting for the host
    // to access our board.  This works because the WAIT request line is also gated with our
    // board CS so the host won't get any wait requests UNTIL it tries to access our board.
    //
    // Then, once we are done processing the host request, we must inactivate the CLKEN line
    // to release the host and allow it to complete it's input/output instruction.  The CLKEN
    // line must remain disabled UNTIL we see that our board CS has become inactive.


    // set the CLKEN line to be an output
    TRISAbits.TRISA5 = 0;
    // and set it to 'enabled' by default
    PORTAbits.RA5 = 1;

#if USE_IF_INTERRUPTS
```

```
    // make sure external interrupt is on default RA2 pin
    INTPPSbits.PORT = PPS_PORTA;
    INTPPSbits.PIN = 2;

    // enable interrupt on RA2/INT rising edge
    INTCONbits.INTEDG = 1;

    PIE0bits.INTE = 1;
#endif

    // configure A0, A1, !IORD as inputs
    TRISCbits.TRISC5 = 1;
    TRISCbits.TRISC6 = 1;
    TRISCbits.TRISC7 = 1;

//************************
// Feature Specific Logic
//************************

    // initialize local variables
    Abort_Command();
}

// Main Loop logic
void RegisterIF_Process(void){

    // if RA2 is active, meaning the host is accessing this board
    if (PORTAbits.RA2){

        // handle the host request
        RegisterIF_Handle_Request();
    }
}

// Handle host request
void RegisterIF_Handle_Request(void){

//************************
// Always Needed Logic
//************************

    unsigned char type = PORTC & INPUT_TYPE_MASK;

    // if the Host is writing data, we need to read in that value being written
    switch (type){
        case INPUT_WR_ADDR0:
        case INPUT_WR_ADDR1:
        case INPUT_WR_ADDR2:

        case INPUT_WR_ADDR3:
            // update the shift chain to read the latest host value written
            ShiftOut();
            break;
    }

//************************
// Feature Specific Logic
//************************

    // Process the host request
```

```
    switch (type){
        case INPUT_RD_ADDR0:                    // Read hard drive status byte
            ShiftOutValues[0] = 0x00;
            break;
        case INPUT_RD_ADDR1:                    // Host reading another data byte
            Process_ReadData();
            break;
        case INPUT_RD_ADDR2:                    // unimplemented
            ShiftOutValues[0] = 0x00;
            break;
        case INPUT_RD_ADDR3:                            // Read Expansion ID register
            ShiftOutValues[0] = IF_FEATURE_HARDDISK;    // indicate this board implements the hard disk feature
            break;

        // Process the host write requests
        case INPUT_WR_ADDR0:
            Process_WriteCmd();                 // Host writing another command byte
            break;
        case INPUT_WR_ADDR1:
            Process_WriteData();                // Host writing another data byte
            break;
        case INPUT_WR_ADDR2:
            WriteRegs[2] = ShiftInValues[0];
            break;
        case INPUT_WR_ADDR3:
            WriteRegs[3] = ShiftInValues[0];
            break;
    }

//************************
// Always Needed Logic
//************************

    // if the Host is reading data, we need to update that value on the read port
    switch (type){
        case INPUT_RD_ADDR0:
        case INPUT_RD_ADDR1:
        case INPUT_RD_ADDR2:
        case INPUT_RD_ADDR3:

            // only update the shift out chain if the value changed from last time
            // this is necessary to prevent the host polling from overwhelming the
            // PIC if this host is reading in a tight loop (worst case scenario)
            // and this logic is running at interrupt level.
            //
            // When this occurs, it was observed that 80uS was spent handling the host
            // IF request, followed by 2.5uS that was available to read the keyboard
            // data stream.  Coincidently, the keyboard clock period is also roughly
            // 80uS, so when thing lined up just right the PIC could see each data
            // bit coming in, but periodically would miss them.
            //
            // By only updating the output shift chain when the output value changes,
            // this drops the time spent on the interface from 80uS to about 59uS
            // and give the keyboard logic more time to sample the bits.  Although
            // an improvement, it was still observed that characters were missed -
            // in fact it looks like it may have made it worse because the free
            // time no longer lines up with the keyboard clock period.
            if (Last_ShiftOut_Value != ShiftOutValues[0]){
                Last_ShiftOut_Value = ShiftOutValues[0];
```

```
                // update the shift chain to reflect the latest read value
                ShiftOut();
            }
            break;
    }

    // Now that the request has been serviced, we need to release the WAIT line to the host
    PORTAbits.RA5 = 0;

    // This is a very short wait.  The logic was changed to remove the wait, and instead interrupt on
    // RA2 falling edge to re-enable the clock line.  There was no observable difference in the timing
    // observed on the scope so we are staying with the polling method which is more solid.
    while (PORTAbits.RA2) ;

    // Now re-enable the clock so our next board access will trigger host wait states
    PORTAbits.RA5 = 1;
}

/*
 * Start inter-byte timeout
 */
void Start_Timeout(){

    IF_Timer = MAX_INTERBYTE_TICKS;
}

/*
 * Stop inter-byte timeout
 */
void Stop_Timeout(){

    // TODO - synchronize with timer interrupt
    IF_Timer = 0x00;
}

/*
 * Timer interrupt callback
 */
void RegisterIF_Timer(void){

    if (IF_Timer > 0){
        // if inter-byte timeout occurred
        if (--IF_Timer == 0){
            Abort_Command();
        }
    }
}

/*
 * Something went wrong processing the host request - clean up and wait for a new command
 */
void Abort_Command(void){


    Stop_Timeout();
    cmdReceived = 0;
    cmdState = CMD_IDLE;
    sector = 0;
    dataBytesRequested = 0;
```

```
    dataBytesProcessed = 0;
}

/*
 * Process a host data request received on the read data port
 */
void Process_ReadData(void){

    // halt timeout while we process this request
    Stop_Timeout();

    switch (cmdState){
        case CMD_RW_PROCESSING_DATA:
            switch (cmdReceived){
                case HDIF_CMD_READ_SECTOR:

                    // give the host the next byte of the sector buffer
                    ShiftOutValues[0] = sectorBuffer[dataBytesProcessed];

                    if (++dataBytesProcessed == dataBytesRequested){
                        // Read Command completed successfully
                        cmdState = CMD_IDLE;
                    }
                    break;

                case HDIF_CMD_GET_DRIVE_STATUS:

                    // give the host the next byte of the driver parameter structure
                    ShiftOutValues[0] = ((UCHAR *) &driveParameters)[dataBytesProcessed];

                    if (++dataBytesProcessed == dataBytesRequested){
                        // Read Command completed successfully
                        cmdState = CMD_IDLE;
                    }
                    break;


                default:
                    // something wrong - read data received without a 'read' command
                    Abort_Command();
                    break;
            }
            break;
        default:
            // something wrong - host is reading more bytes than it requested
            cmdState = CMD_IDLE;
            // force data read to be zeroes
            ShiftOutValues[0] = DUMMY_READ_DATA;
            break;
    }

    if (cmdState != CMD_IDLE){
        // Reset timeout as read data process is not yet complete
        Start_Timeout();
    }
}

/*
 * Process a new byte of host data received on the write command port
```

```
 */
void Process_WriteCmd(void){

    // halt timeout while we process this request
    Stop_Timeout();

    switch (cmdState){
        case CMD_IDLE:
            cmdReceived = ShiftInValues[0];
            dataBytesProcessed = 0;

            switch (cmdReceived){
                case HDIF_CMD_READ_SECTOR:
                case HDIF_CMD_WRITE_SECTOR:
                case HDIF_CMD_FORMAT_DEVICE:
                    cmdState = CMD_RW_LSB_SECTOR;
                    break;
                case HDIF_CMD_GET_DRIVE_STATUS:
                    dataBytesRequested = sizeof(DriveParameters_T);

                    cmdState = CMD_RW_PROCESSING_DATA;
                    break;
                default:
                    // unsupported command - keep looking for a valid command
                    break;
            }
            break;

        // READ/WRITE command states
        case CMD_RW_LSB_SECTOR:
            sector = ShiftInValues[0];
            cmdState = CMD_RW_MSB_SECTOR;
            break;
        case CMD_RW_MSB_SECTOR:
            sector += (ShiftInValues[0] * 0x100);

            if (cmdReceived == HDIF_CMD_FORMAT_DEVICE){
                // full device format command received
                if (sector != HDIF_FORMAT_KEY){
                    // sector must be supported, otherwise invalid command
                    Abort_Command();
                }else{
                    // re-map the SPI logic to connect with the Flash chip
                    Map_SPI_To_Flash();
                    // process the format command
                    Format_Logical_Drive();
                    // re-map the SPI logic back to the host Register IF
                    Map_SPI_To_IO();

                    cmdState = CMD_IDLE;
                }
            }else{
                // continue getting command structure for other longer commands
                cmdState = CMD_RW_LSB_NUMBYTES;
            }
            break;
        case CMD_RW_LSB_NUMBYTES:
            dataBytesRequested = ShiftInValues[0];
            cmdState = CMD_RW_MSB_NUMBYTES;
            break;
```

```
        case CMD_RW_MSB_NUMBYTES:
            dataBytesRequested += (ShiftInValues[0] * 0x100);

            // validate the hosts request
            if (sector >= HD_NUM_LOGICAL_SECTORS){
                Abort_Command();
            }else{
                if ((dataBytesRequested) > LOGICAL_SECTOR_SIZE){
                    dataBytesRequested = LOGICAL_SECTOR_SIZE;
                }

                if (cmdReceived == HDIF_CMD_READ_SECTOR){

                    // re-map the SPI logic to connect with the Flash chip
                    Map_SPI_To_Flash();
                    // read the specified sector into our buffer
                    Read_Logical_Sector(sector, &sectorBuffer[0]);
                    // re-map the SPI logic back to the host Register IF
                    Map_SPI_To_IO();
                }

                cmdState = CMD_RW_PROCESSING_DATA;
            }
            break;

        default:
        case CMD_RW_PROCESSING_DATA:
            // something wrong - host is sending us too much command information
            Abort_Command();
            break;
    }

    if (cmdState != CMD_IDLE){
        // Reset timeout as command is not yet complete
        Start_Timeout();
    }
}

/*
 * Process a new byte of host data received on the write data port
 */
void Process_WriteData(void){

    unsigned long int address;

    // halt timeout while we process this request
    Stop_Timeout();

    switch (cmdState){
        case CMD_RW_PROCESSING_DATA:
            switch (cmdReceived){
                case HDIF_CMD_WRITE_SECTOR:
                    // store the byte received from the host in the buffer
                    sectorBuffer[dataBytesProcessed++] = ShiftInValues[0];

                    // if all bytes received
                    if (dataBytesProcessed == dataBytesRequested){
```

```
                        // re-map the SPI logic to connect with the Flash chip
                        Map_SPI_To_Flash();
                        // write the host sector data to the specified sector
                        Write_Logical_Sector(sector, &sectorBuffer[0]);
                        // re-map the SPI logic back to the host Register IF
                        Map_SPI_To_IO();

                        // Write Command completed successfully
                        cmdState = CMD_IDLE;
                    }
                    break;

                default:
                    // something wrong - write data received without a 'write' command
                    Abort_Command();
                    break;
            }
            break;
        default:
            // something wrong - host is sending us too much command information
            Abort_Command();
            break;
    }

    if (cmdState != CMD_IDLE){
        // Reset timeout as write data process is not yet complete
        Start_Timeout();
    }
}
```

```
#include <xc.h>
#include <pic16f15244.h>
#include <stdbool.h>
#include "ShiftOut.h"
#include "RemapPins.h"
#include "Utility.h"

// memory images of shifter chain values
unsigned char ShiftOutValues[1];
unsigned char ShiftInValues[1];
// first bit read in during the input shift chain load process, before SPI takes over
bool FirstBitRead;

// Remap the functions pins for the shifter hardware to use the chip's SPI features
void ShiftOut_RemapPins(void){

    // Setting the tristate values for the SPI pins is important and must be taken care of manually here
    // In this case SHIFT_DATA_OUT and SHIFT_CLK are SPI features, OUT_LATCH_CLK and IN_LATCH_MODE are driven in software
    TRISCbits.TRISC0 = 0;
    TRISCbits.TRISC1 = 0;
    TRISCbits.TRISC2 = 0;
    TRISCbits.TRISC4 = 0;

    // and set the output pins to low level
    PORTC &= 0xe8;

    // Set the SHIFT_DATA_IN pin to be an input
    TRISCbits.TRISC3 = 1;
    // Make sure input is using Schmitt Trigger CMOS input levels
    INLVLCbits.INLVLC3 = 1;

    // This implementation uses the hardware SPI port feature of the PIC
    // In order to do this, the SPI functions need to be mapped to the
    // port pins actually connected to the shifter chain.
    RC1PPS = PPS_SCL1SCK1;
    RC0PPS = PPS_SDA1SDO1;

    // Mapping for 'SDI' input
    SSP1DATPPSbits.PORT = PPS_PORTC;
    SSP1DATPPSbits.PIN = SHIFT_DATA_IN_PORT_PIN;


    // Important! This is very non-intuitive.  If you move the SCLK output pin assignment
    // you MUST ALSO move the SCLK input pin assignment.
    //
    // the clock for shifting data out and shifting data in are separate inside the
    // chip, but normal usage would dictate that they be connected to the same pin.
    // Failure to do this will usually result in NO DATA SHIFTED IN.
    SSP1CLKPPSbits.PORT = PPS_PORTC;
    SSP1CLKPPSbits.PIN = SHIFT_CLK_PORT_PIN;
}


// Initialize the shifter logic
void ShiftOut_Init(void){

    unsigned char temp;

    ShiftOut_RemapPins();       // re-map the uP functions pins so the SPI features map to the shift chain

    SSPEN = 0;                  // disable SSP for configuration

    SSP1CON1bits.WCOL = 0;      // clear any write collision flag
    SSP1CON1bits.SSPOV = 0;     // clear any read overflow flag
    SSP1CON1bits.CKP = 1;       // Idle state for clock is low
```

```
    SSP1CON1bits.SSPM = 0;      // Select SPI Master mode, with Clock = Fosc / 4

    SSP1CON3bits.BOEN = 1;      // (0) if Rx overrun, set SSPOV and don't overwrite SSP1BUF
                                // (1) if Rx overrun, set SSPOV and overwrite SSP1BUF

    SSP1STATbits.SMP = 0;       // Input data is sampled at the middle (0) (vs end (1)) of data output time
    SSP1STATbits.CKE = 0;       // Tx occurs on idle->active clock (0) (vs active->idle (1))

    SSPEN = 1;                  // enable SSP operation

    temp = SSP1BUF;             // read any byte to clear the buffer full flag

    SSP1IE = 0;                 // disable SSP interrupts

    // initialize the output shift chain values
    ShiftOutValues[0] = 0xAA;

    // update the shift register chain with these new values
    ShiftOut();
}

// physically drive the specified value out to the physical hardware
void ShiftOut(void){

    //************************************************************
    // before we start shifting, we must transfer the input values into
    // the input shift register chain.  This is more complicated than you
    // might expect as the 74166 shift/!load pin ALSO requires a rising edge
    // on the clock pin WHILE shift/!load is low.  So we have to temporarily
    // remap the clock pin to be a general purpose pin (vs the clock for the
    // SPI serial port), issue one clock pulse, and then re-map the pin
    // back to being the clock pin for the SPI serial port.
    //************************************************************


    PORTCbits.RC1 = 0;          // set clock low
    RC1PPS = PPS_LATxy;         // remap the RC1 pin to use the latch feature
    PORTCbits.RC4 = 0;          // set mode = load
    PORTCbits.RC1 = 1;          // set clock high
    PORTCbits.RC4 = 1;          // set mode = shift

    // now read in, and save, the first data bit that was just clocked out
    RC3PPS = PPS_LATxy;                 // remap the RC3 pin to use the latch feature
    FirstBitRead = PORTCbits.RC3;       // read in the first bit
    SSP1DATPPSbits.PORT = PPS_PORTC;    // Re-map RC3 for 'SDI' input
    SSP1DATPPSbits.PIN = SHIFT_DATA_IN_PORT_PIN;

    PORTCbits.RC1 = 0;          // set clock low

    RC1PPS = PPS_SCL1SCK1;      // remap the RC1 pin to use the SPI clock feature


    //************************************************************
    // The Data has now been loaded into the shift register chain
    // and it is now time to serially clock it into the processor
    // using the SPI hardware.
    //************************************************************



    for (unsigned int x=0; x<NUM_ENTRIES(ShiftOutValues); x++){

        unsigned int timeout = 0;
```

```
        // shift out the next MSB byte
        SSP1BUF = ShiftOutValues[NUM_ENTRIES(ShiftOutValues) - 1 - x];

        // wait until the byte has been fully sent
        while ((SSP1STATbits.BF == 0) && (timeout < 30000)) timeout++;

        // read the input shift byte and store it in the next LSB byte
        ShiftInValues[x] = SSP1BUF;
    }


    //************************************************************
    // The entire shift chain has been updated so now we have to
    // transfer the output shift register values into the output latches
    //************************************************************


    PORTCbits.RC2 = 1;
    PORTCbits.RC2 = 0;


    //************************************************************
    // now the input values need to be shifted over one bit and the bit read in
    // during the shifter load process needs to be re-inserted in it's proper place.
    // FirstBitRead >> ShiftInValues[]
    //************************************************************

    bool shiftInBit = FirstBitRead;
    for (int x=0; x<NUM_ENTRIES(ShiftInValues); x++){
        bool shiftOutBit = ShiftInValues[x] & 0x01;
        ShiftInValues[x] = ((ShiftInValues[x] >> 1) & 0x7f) | (shiftInBit ? 0x80 : 0x00);
        shiftInBit = shiftOutBit;
    }
}
```

```
/*
 * File:   SPIMemory.c
 * Author: Alec
 *
 * Created on September 14, 2021, 8:35 AM
 *
 * SPI Flash is wired in as follows:
 *              -----------
 * RB5 --> (1) | !CS   VCC | (8) +5V
 * RB7 <-- (2) | DO  !HOLD | (7) +5V
 *     +5V (3) | !WP   CLK | (6) <-- RC1  {FROM COMMON SHIFT CHAIN SPI CLK}
 *     GND (4) | GND    DI | (5) <-- RC0  {FROM COMMON SHIFT CHAIN SPI DATA OUT}
 *              -----------
 *
 * SPI RAM is wired in as follows:
 *              -----------
 * RB4 --> (1) | !CS   VCC | (8) +5V
 * RB6 <-- (2) | DO  !HOLD | (7) <-- +5V
 *         (3) | N/U   CLK | (6) <-- RC1  {FROM COMMON SHIFT CHAIN SPI CLK}
 *     GND (4) | GND    DI | (5) <-- RC0  {FROM COMMON SHIFT CHAIN SPI DATA OUT}
 *              -----------
 */

#include <pic16f15244.h>
#include <stdbool.h>

#include "SPIMemory.h"
#include "RemapPins.h"
#include "ShiftOut.h"
#include "LogicalDrv.h"

// The following assume that all SPI flash pins are configured on PORTB
// NOTE: not all code uses these, if you move the feature you still have to modify code
//       (this was done to keep code faster and more readable)
#define SPIFLASH_DATA_OUT_PORT_PIN  7   /* Flash data out is on RB7 */
#define SPIRAM_DATA_OUT_PORT_PIN    6   /* RAM data out is on RB6 */
#define SPIFLASH_CS_PORT_PIN        5   /* Flash !CS is on RB5 */
#define SPIRAM_CS_PORT_PIN          4   /* RAM !CS is on RB4 */

// The following are SPI flash pins that share the standard SHIFTOUT.C SPI port lines
#define SPIF_CLK_PORT_PIN       1   /* clock is on RC1 */
#define SPIF_DATA_IN_PORT_PIN   0   /* data in is on RC0 */

void Set_CS(bool active);

// which device is currently selected
enum {
    RegisterIF_Selected = 0x00,
    Flash_Selected,
    RAM_Selected
} SPI_Selected;
```

```
void SPIMemory_Init(void){

    TRISBbits.TRISB7 = 1;   // RB7 is an input
    TRISBbits.TRISB6 = 1;   // RB6 is an output
    TRISBbits.TRISB5 = 0;   // RB5 is an output
    TRISBbits.TRISB4 = 0;   // RB4 is an output

    SPI_Selected = RegisterIF_Selected;

    Set_CS(false);
}

/* Set_CS
 *
 * enable/disable the currently selected SPI device
 */
void Set_CS(bool active){

    switch (SPI_Selected){
        default:
        case RegisterIF_Selected:
            PORTBbits.RB4 = 1;  // make sure RAM !CS is OFF
            PORTBbits.RB5 = 1;  // make sure Flash !CS is OFF
            break;
        case Flash_Selected:
            PORTBbits.RB4 = 1;          // make sure RAM !CS is OFF
            PORTBbits.RB5 = !active;    // set Flash !CS as requested
            break;
        case RAM_Selected:
            PORTBbits.RB5 = 1;          // make sure Flash !CS is OFF
            PORTBbits.RB4 = !active;    // set RAM !CS as requested
            break;
    }
}

/* Map_SPI_To_Flash
 *
 * When using the flash, the SPI data in feature needs to be re-mapped to the pin connected
 * to the flash data out pin.
 */
void Map_SPI_To_Flash(void){

    PORTBbits.RB4 = 1;  // make sure RAM !CS is OFF

    SPI_Selected = Flash_Selected;

    // Map the SPI 'SDI' input function to the pin connected to the Flash data out pin
    SSP1DATPPSbits.PORT = PPS_PORTB;
    SSP1DATPPSbits.PIN = SPIFLASH_DATA_OUT_PORT_PIN;
}

/* Map_SPI_To_RAM
```

```
 *
 * When using the RAM, the SPI data in feature needs to be re-mapped to the pin connected
 * to the RAM data out pin.
 */
void Map_SPI_To_RAM(void){

    PORTBbits.RB5 = 1;  // make sure Flash !CS is OFF

    SPI_Selected = RAM_Selected;

    // Map the SPI 'SDI' input function to the pin connected to the RAM data out pin
    SSP1DATPPSbits.PORT = PPS_PORTB;
    SSP1DATPPSbits.PIN = SPIRAM_DATA_OUT_PORT_PIN;
}

/* Map_SPI_To_IO
 *
 * When not using the flash/ram, the SPI data in feature needs to be re-mapped to the pin connected
 * to the I/O chain data out pin.
 */
void Map_SPI_To_IO(void){

    PORTBbits.RB4 = 1;  // make sure RAM !CS is OFF
    PORTBbits.RB5 = 1;  // make sure Flash !CS is OFF

    SPI_Selected = RegisterIF_Selected;

    // Map the SPI 'SDI' input function to the pin connected to the I/O Shift Chain data out pin
    SSP1DATPPSbits.PORT = PPS_PORTC;
    SSP1DATPPSbits.PIN = SHIFT_DATA_IN_PORT_PIN;
}

/* SPI_Write_Enable
 *
 * enable/disable writing to the currently selected SPI device
 */
void SPI_Write_Enable(bool enable){

    switch (SPI_Selected){
        default:
        case RAM_Selected:
        case RegisterIF_Selected:
            break;
        case Flash_Selected:
            if (SPI_Write_Enabled() && enable) return;
            if (!SPI_Write_Enabled() && !enable) return;

            Set_CS(true);
            SSP1BUF = enable ? WRITE_ENABLE : WRITE_DISABLE;
            while (SSP1STATbits.BF == 0) ;
            Set_CS(false);
            break;
    }
```

```
    }
}

/* SPI_Write_Enabled
 *
 * return true if currently selected SPI device is enabled for writing
 */
bool SPI_Write_Enabled(void){

    bool writeEnabled = true;

    switch (SPI_Selected){
        default:
        case RAM_Selected:
        case RegisterIF_Selected:
            break;
        case Flash_Selected:
            writeEnabled = (Read_SPI_Register(1) & WEL);
            break;
    }

    return writeEnabled;
}


/* Get_SPI_Device_ID
 *
 * return the currently selected SPI device's device ID
 */
UINT16 Get_SPI_Device_ID(void){

    UINT16 retCode;

    Set_CS(true);

    SSP1BUF = MFGR_DEVICE_ID;
    while (SSP1STATbits.BF == 0) ;

    SSP1BUF = DUMMY_COMMAND;
    while (SSP1STATbits.BF == 0) ;
    SSP1BUF = DUMMY_COMMAND;
    while (SSP1STATbits.BF == 0) ;
    SSP1BUF = DUMMY_COMMAND;
    while (SSP1STATbits.BF == 0) ;

    SSP1BUF = DUMMY_COMMAND;
    while (SSP1STATbits.BF == 0) ;
    retCode = SSP1BUF * 0x100;          // get manufacturer code into high bits


    SSP1BUF = DUMMY_COMMAND;
    while (SSP1STATbits.BF == 0) ;
    retCode += SSP1BUF;                 // get device ID into low bits
```

```
    Set_CS(false);

    return retCode;
}

/* SPI_Device_Busy
 *
 * return true if currently selected SPI device is busy and needs more time
 */
bool SPI_Device_Busy(void){

    bool busy = false;

    switch (SPI_Selected){
        default:
        case RAM_Selected:
        case RegisterIF_Selected:
            break;
        case Flash_Selected:
            busy = (Read_SPI_Register(1) & BUSY);
            break;
    }

    return busy;
}

/* Read_SPI_Register (on currently selected SPI device)
 *
 * which  - SPI internal device register address (1, 2, 3, etc...)
 */
UCHAR Read_SPI_Register(UCHAR which){

    UCHAR retCode;

    Set_CS(true);

    switch (which){
        default:
        case 1:
            SSP1BUF = READ_STATUS_REG1;
            break;
        case 2:
            SSP1BUF = READ_STATUS_REG2;
            break;
        case 3:
            SSP1BUF = READ_STATUS_REG3;
            break;
    }
    while (SSP1STATbits.BF == 0) ;

    SSP1BUF = DUMMY_COMMAND;
```

```
    while (SSP1STATbits.BF == 0) ;
    retCode = SSP1BUF;                  // get register contents

    Set_CS(false);

    return retCode;
}

/* Write_SPI_Register (on currently selected SPI device)
 *
 * which  - SPI internal device register address (1, 2, 3, etc...)
 * value  - value to write
 */
void Write_SPI_Register(UCHAR which, UCHAR value){

    Set_CS(true);

    switch (which){
        default:
        case 1:
            SSP1BUF = WRITE_STATUS_REG1;
            break;
        case 2:
            SSP1BUF = WRITE_STATUS_REG2;
            break;
        case 3:
            SSP1BUF = WRITE_STATUS_REG3;
            break;
    }
    while (SSP1STATbits.BF == 0) ;

    SSP1BUF = value;
    while (SSP1STATbits.BF == 0) ;

    Set_CS(false);
}

/* Read_SPI_Data (from currently selected SPI device)
 *
 * from   - SPI internal device address to start reading
 * length - #of bytes to read
 * buffer - pointer to buffer to store data read
 */
void Read_SPI_Data(SPI_ADDRESS_TYPE from, UINT16 length, UCHAR *buffer){

    UCHAR AB0, AB1, AB2;

    // wait until any prior write operations have completed
    while (SPI_Device_Busy()) ;

    AB0 = (from & 0x000000ff);
    from /= 0x100;
    AB1 = (from & 0x000000ff);
```

```
    AB1 = (from & 0x0000ff00);
    from /= 0x100;
    AB2 = (from & 0x000000ff);

    Set_CS(true);

    SSP1BUF = READ_DATA;
    while (SSP1STATbits.BF == 0) ;

    // flash access use 24 bit addressing
    // ram access uses 16 bit addressing
    if (SPI_Selected == Flash_Selected){
        SSP1BUF = AB2;
        while (SSP1STATbits.BF == 0) ;
    }
    SSP1BUF = AB1;
    while (SSP1STATbits.BF == 0) ;
    SSP1BUF = AB0;
    while (SSP1STATbits.BF == 0) ;

    for ( ;(length != 0); buffer++, length--){
        SSP1BUF = DUMMY_COMMAND;
        while (SSP1STATbits.BF == 0) ;
        *buffer = SSP1BUF;
    }

    Set_CS(false);
}


/* Write_SPI_Data (to currently selected SPI device)
 *
 * from   - SPI internal device address to start writing
 * length - #of bytes to write
 * buffer - pointer to buffer to use as data source
 */
void Write_SPI_Data(SPI_ADDRESS_TYPE from, UINT16 length, UCHAR *buffer){

    UCHAR AB0, AB1, AB2;
    UINT16 pLen, wLen;

    // wait until any prior write operations have completed
    while (SPI_Device_Busy()) ;

    // ensure that write is enabled
    SPI_Write_Enable(true);

    AB0 = (from & 0x000000ff);
    from /= 0x100;
    AB1 = (from & 0x000000ff);
    from /= 0x100;
    AB2 = (from & 0x000000ff);
```

```
    Set_CS(true);

    switch (SPI_Selected){
        case RegisterIF_Selected:

            // this logic is not intended to drive the Register I/F
            break;

        case RAM_Selected:
            SSP1BUF = PAGE_PROGRAM;
            while (SSP1STATbits.BF == 0) ;

            SSP1BUF = AB1;
            while (SSP1STATbits.BF == 0) ;
            SSP1BUF = AB0;
            while (SSP1STATbits.BF == 0) ;

            for ( ;(length != 0); buffer++, length--){
                SSP1BUF = *buffer;
                while (SSP1STATbits.BF == 0) ;
            }

            Set_CS(false);
            break;

        // The flash has an extra requirement during writes in that only a page (256 bytes)
        // can be written at a time.  A page is defined to start on addresses where A7:A0 = 0x00
        // and runs for 256 bytes.  Attempting to program across the upper page boundary
        // will result in a wrap around to the beginning of the page.  Consequently, any writes
        // that cross page boundaries must be broken up into multiple smaller page size writes.
        // Also remember that the Write Enable Latch is cleared after every page program operation.
        case Flash_Selected:


            // calculate how many bytes will fit into the first page
            // based upon the LSB starting address AB0
            pLen = FLASH_PAGE_SIZE - AB0;

            while (length != 0){

                // determine #of bytes to write to this page
                wLen = (pLen >= length) ? length : pLen;

                // initiate a page write at address AB2:AB1:AB0 in the flash
                SSP1BUF = PAGE_PROGRAM;
                while (SSP1STATbits.BF == 0) ;


                SSP1BUF = AB2;
                while (SSP1STATbits.BF == 0) ;
                SSP1BUF = AB1;
                while (SSP1STATbits.BF == 0) ;
                SSP1BUF = AB0;
```

```
                while (SSP1STATbits.BF == 0) ;

                // write out 'wLen' bytes
                for (UINT16 x=wLen;(x != 0); buffer++, x--){
                    SSP1BUF = *buffer;
                    while (SSP1STATbits.BF == 0) ;
                }

                // disable chip select to start the internal flash write cycle
                Set_CS(false);

                // adjust overall remaining length by the #of bytes just written
                length -= wLen;

                if (length > 0){
                    // subsequent pages can write a full page
                    pLen = FLASH_PAGE_SIZE;

                    // update SPI address bytes for the data just written
                    AB0 = 0x00;
                    AB1++;
                    if (AB1 == 0x00) AB2++;

                    // wait until this page write completes
                    while (SPI_Device_Busy()) ;

                    // re-enable the Write Enable Latch which is cleared after
                    // every page program operation
                    SPI_Write_Enable(true);

                    // re-enable the chip select for the next page write cycle
                    Set_CS(true);
                }
            }
            break;

    }
}

/* Erase_SPI_Sector
 *
 * erase 4K sector containing the specified SPI internal device address
 */
void Erase_SPI_Sector(SPI_ADDRESS_TYPE address){

    UCHAR AB0, AB1, AB2;

    switch (SPI_Selected){
        default:
        case RAM_Selected:
        case RegisterIF_Selected:
            break;
        case Flash_Selected:
```

```
            // wait until any prior write operations have completed
            while (SPI_Device_Busy()) ;

            // ensure that write is enabled
            SPI_Write_Enable(true);

            AB0 = (address & 0x000000ff);
            address /= 0x100;
            AB1 = (address & 0x000000ff);
            address /= 0x100;
            AB2 = (address & 0x000000ff);

            Set_CS(true);

            SSP1BUF = SECTOR_ERASE_4K;
            while (SSP1STATbits.BF == 0) ;

            SSP1BUF = AB2;
            while (SSP1STATbits.BF == 0) ;
            SSP1BUF = AB1;
            while (SSP1STATbits.BF == 0) ;
            SSP1BUF = AB0;
            while (SSP1STATbits.BF == 0) ;


            Set_CS(false);

            // wait until the erase operation has completed
            while (SPI_Device_Busy()) ;

            break;
    }
}

/* Erase_SPI_Chip
 *
 * erase the ENTIRE chip
 */
void Erase_SPI_Chip(void){

    switch (SPI_Selected){
        default:
        case RAM_Selected:
        case RegisterIF_Selected:
            break;
        case Flash_Selected:

            // wait until any prior write operations have completed
            while (SPI_Device_Busy()) ;

            // ensure that write is enabled
            SPI_Write_Enable(true);
```

```
            Set_CS(true);

            SSP1BUF = CHIP_ERASE1;
            while (SSP1STATbits.BF == 0) ;

            Set_CS(false);

            // wait until the erase process has completed
            while (SPI_Device_Busy()) ;

            break;
    }
}

/* SPI_RAM_Test
 *
 * check that the SPI RAM is working
 */
bool SPI_RAM_Test(void){

    SPI_ADDRESS_TYPE address;
    bool success = true;

    for (address=0; (address<SPI_RAM_SIZE) && success; address+=NUM_ENTRIES(sectorBuffer)){

        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++) sectorBuffer[x] = 0xaa;
        Write_SPI_Data(address, NUM_ENTRIES(sectorBuffer), sectorBuffer);
        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++) sectorBuffer[x] = 0x00;
        Read_SPI_Data(address, NUM_ENTRIES(sectorBuffer), sectorBuffer);
        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++){
            if (sectorBuffer[x] != 0xaa) success = false;
        }

        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++) sectorBuffer[x] = 0x55;
        Write_SPI_Data(address, NUM_ENTRIES(sectorBuffer), sectorBuffer);
        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++) sectorBuffer[x] = 0x00;
        Read_SPI_Data(address, NUM_ENTRIES(sectorBuffer), sectorBuffer);
        for (int x=0; x<NUM_ENTRIES(sectorBuffer); x++){
            if (sectorBuffer[x] != 0x55) success = false;
        }
    }

    return success;
}
```

```
#include <xc.h>
#include "Timer.h"
#include "RegisterIF.h"

// DIAGNOSTICS ONLY - enable to check the Timer0 period using the precision timer
#define TEST_PRECISION_TIMER    0

// Timer Initialization
void Timer_Init(void){

    // Initialize Timer0 to interrupt every 65mS
    // 1MHz / 256 (prescaler) / 1 (postscaler) / 255 (TMR0H) = 15.32Hz (65.28mS)
    // REMEMBER to update Timer0uSPeriod() if you change this
    TMR0 = 0;                   // clear timer so a full cycle can occur
    T0CON0bits.MD16 = 0;        // mode (0=8 bit, 1=16 bit)
    T0CON0bits.OUTPS = 0x00;    // 1:1 post-scaler
    T0CON1bits.CS = 3;          // HFINTOSC (see config bits, usually RSTOSC = HFINTOSC_1MHZ)
    T0CON1bits.CKPS = 8;        // 1:256 pre-scaler
    TMR0H = 0xff;               // count up to 255
    T0CON0bits.EN = 1;          // enable timer
    PIE0bits.TMR0IE = 1;        // enable interrupts for timer 0
    PIR0bits.TMR0IF = 0;        // clear any pending interrupt condition

    // Initialize Timer1 to be a free running 16bit hardware 8uS counter
    // 1MHz / 8 (prescaler) = 125KHz (8uS)
    // REMEMBER to update PrecisionTimeruSPeriod() if you change this
    T1CLK = 3;                  // HFINTOSC (see config bits, usually RSTOSC = HFINTOSC_1MHZ)
    T1CONbits.CKPS = 3;         // 1:8 prescaler
    T1CONbits.RD16 = 1;         // enable 16bit r/w operations
    T1CONbits.TMR1ON = 1;       // enable timer1
}


// get Timer0 uS period
inline const unsigned short int Timer0uSPeriod(void){

    return TIMER0USPERIOD;
}

// Timer 0 interrupt handler
void Timer0_Handler(void){

   // only process Timer0-triggered interrupts
   if (PIE0bits.TMR0IE && PIR0bits.TMR0IF) {

       RegisterIF_Timer();

       // clear this interrupt condition
```

```
        PIR0bits.TMR0IF = 0;
    }
}
```