```
/*
 * File:   Common.h
 * Author: Alec
 *
 * Created on October 15, 2021, 8:35 AM
 */

#ifndef COMMON_H
#defineCOMMON_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Utility.h"
#include <stdbool.h>

#ifdef__cplusplus
}
#endif
```

```
#endif/* COMMON_H */
```

```
/*
 * File:   FDC_Interface.h
 * Author: Alec
 *
 * Created on October 22, 2021, 7:52 AM
 */

#ifndef FDC_INTERFACE_H
#defineFDC_INTERFACE_H

#ifdef __cplusplus
extern "C" {
#endif

#include "Common.h"
#include "FDC_Logic.h"

/* CheckDisk() was originally written as a means to debug the drive read/write
 * logic without all of the overly complicated format requirements.  This is not
 * necessary for normal operation and is only left here as a means to debug
 * complicated situations that may arise in the future.  Eventually this
 * could be removed.
 */
#define ENABLE_CHECKDISK_LOGIC  0

typedef enum{
    STATUS_SUCCESS = 0,
    STATUS_CANT_FIND_SECTOR,
    STATUS_CANT_FIND_DATA_MARK,
    STATUS_DELETED_DATA_MARK,
    STATUS_INVALID_DATA_MARK,
    STATUS_INVALID_CRC,
    STATUS_WRITE_FAULT,
    STATUS_UNKNOWN_ERROR
} READ_WRITE_STATUS_TYPE;

extern void Detect_Drives(void);
extern bool Drive_Select(DRIVE_SELECTED_TYPE drive);
extern bool Drive_Motor_At_Full_Speed(void);
extern READ_WRITE_STATUS_TYPE Read_Sector(TRACK_TYPE track, SECTOR_TYPE sector, SECTOR_BUFFER_TYPE buffer);
extern READ_WRITE_STATUS_TYPE Write_Sector(TRACK_TYPE track, SECTOR_TYPE sector, SECTOR_BUFFER_TYPE buffer);
extern bool Format_Drive(void);
#if ENABLE_CHECKDISK_LOGIC
extern bool CheckDisk(UCHAR drive);
#endif


#ifdef __cplusplus
}
#endif

#endif/* FDC_INTERFACE_H */
```

```
/*
 * File:   FDC_Logic.h
 * Author: Alec
 *
 * Created on October 22, 2021, 7:52 AM
 */

#ifndef FDC_LOGIC_H
#defineFDC_LOGIC_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Common.h"
#include "FDC_Physical.h"
#include "Timer.h"

#define NUM_DRIVES_SUPPORTED    4

#define LAST_OUTER_TRACK        43          /* tracks 0-LAST_OUTER_TRACK use low current writes, other tracks use high current write */
#define HEAD_LOAD_DELAYMS       40          /* Min delay after loading head before R/W can commence */
#define STEP_DELAYMS            10          /* Min delay after stepping the head to a new track */

#define NUM_TRACKS_PER_DRIVE    77
    #define FIRST_TRACK_INDEX       0
    #define NO_TRACK_SELECTED       0xff    /* track value used when the current drive track is unknown */
#define NUM_SECTORS_PER_TRACK   26
    #define FIRST_SECTOR_INDEX      1
#define NUM_BYTES_PER_SECTOR    128
    #define MIN_SECTOR_SIZE         4
#define NUM_READ_RETRIES        3

#define DRIVE_RPM               360

// Max revolutions to check before declaring an error that sector ID record can't be found
// (use 4 to guarantee at least 3 complete revolutions)
#define MAX_REVOLUTIONS_TO_FIND_SECTOR_LIMIT 4

/*****************************************************************************
 *
 * Why are there 'gap' bytes?
 *
 * The purpose of the gap bytes is not to store some specific value on the disk,
 * but rather to 'occupy' space before or after important data.
 *
 * Gap bytes before meaningful data serve to have known good data before an
 * address mark so that the data converter can be in a known good state before
 * the address mark is read.
 *
 * Gap bytes after meaningful data serve as a guarantee that the last meaningful
 * data byte was completely written to the disk.  Terminating a disk write after
 * the last data byte has been written, often leaves the last byte in a 'partially'
 * written state which causes it to be read as garbage later.
 *
 * When writing, the gap bytes before allow a margin of error when turning on
 * the head write current so that it doesn't accidentally overwrite good data
 * located immediately before the write area.
 *
 *****************************************************************************/

// Gaps start out with one value and end with another
#define GAP_START_DATA_VALUE                0xff
#define GAP_END_DATA_VALUE                  0x00
#define INDEX_ADDRESS_MARK_DATA_VALUE       0xfc
#define ID_ADDRESS_MARK_DATA_VALUE          0xfe
#define DATA_MARK_DATA_VALUE                0xfb
#define DELETED_DATA_MARK_DATA_VALUE        0xf8
#define FORMATTED_DATA_VALUE                0xe5

#define POST_INDEX_HOLE_GAP_START_BYTES     40
```

```
#define POST_INDEX_HOLE_GAP_END_BYTES       6
#define POST_INDEX_MARK_GAP_START_BYTES     26
#define POST_INDEX_MARK_GAP_END_BYTES       6
#define POST_ID_RECORD_GAP_START_BYTES      11
#define POST_ID_RECORD_GAP_END_BYTES        6
#define POST_DATA_RECORD_GAP_START_BYTES    27
#define POST_DATA_RECORD_GAP_END_BYTES      6

typedef enum{
    ClockNormal = CBN,
    ClockIndex = CBI,
    ClockID = CBD
} FDC_CLOCK_TYPE;

typedef UCHAR TRACK_TYPE;
typedef UCHAR SECTOR_TYPE;
typedef UCHAR SECTOR_BUFFER_TYPE[NUM_BYTES_PER_SECTOR];


typedef struct{
    bool present;           /* drive is connected and at least partially functional */
    bool headLoaded;        /* current 'head loaded' state */
    TRACK_TYPE track;       /* current 'track' state */
    bool readWriteWorked;   /* set if a read/write sequence on this drive has been successful */
} DRIVE_TYPE;

typedef enum{
    DriveA = 0,
    DriveB,
    DriveC,
    DriveD,
    NoDriveSelected = 255
} DRIVE_SELECTED_TYPE;

extern DRIVE_SELECTED_TYPE Drive_Selected;
extern DRIVE_TYPE Drive[NUM_DRIVES_SUPPORTED];

extern void uPD372D_Logic_Init(void);
extern bool DriveSelect(DRIVE_SELECTED_TYPE drive);
extern void DelayMs(UCHAR count);
extern bool Seek(UCHAR track);
extern bool FindTrackZero(void);
extern bool StepHeadIn(bool inDirection);
extern void Load_Head(bool load);
extern void Write_Gap(UCHAR startCount, UCHAR endCount, FDC_CLOCK_TYPE clockType);
extern bool Locate_Sector(TRACK_TYPE track, SECTOR_TYPE sector);


#ifdef __cplusplus
}
#endif

#endif/* FDC_LOGIC_H */
```

```
/*
 * File:   FDC_Physical.h
 * Author: Alec
 *
 * Created on October 7, 2021, 6:08 PM
 */

#ifndef FDC_PHYSICAL_H
#defineFDC_PHYSICAL_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Common.h"

/* The FDC address lines come out of the PIC PortA lines, however they are not
 * in consecutive order.  Since speed is critical here they are being pre-mapped
 * to these enum values.
 *
 * A0 is on RA2, A1 is on RA4, A2 is on RA5
 */
typedef enum{
    REG0 = 0x00,
    REG1 = 0x04,
    REG2 = 0x10,
    REG3 = 0x14,
    REG4 = 0x20,
    REG5 = 0x24,
    REG6 = 0x30,
    REG7 = 0x34,
} FDC_REG_ADDR_TYPE;


#define UPD372D_BASE    0xf0

/*
 * Basic operation during write is as follows:
 *   a) setup WR1 to the required clock type for the next byte(s)
 *   b) setup WR2 to the required data value for the next byte(s)
 *   c) setup WR3 to initiate the write command (STT=1 starts the command)
 *
 * the data/clock type is loaded into the FDC state machine at the moment STT becomes 1
 *   (and not at the time WR1/WR2 are set)
 * if new data/clock type is not loaded before the state machine starts writing the next
 * byte, it continues to use the same data/clock type.
 */

#define UPD372D_WR0     REG0
    typedef enum{
        WFR = 0x02,     /* Write Fault Reset - set high for 10uS to reset a drive write fault condition */
        LCT = 0x04,     /* Low Current - must be '1' for writes to outer tracks (0-LAST_OUTER_TRACK), and '0' for inner tracks (LAST_OUTER_TRACK-76) */
        HLD = 0x08,     /* Head LoaD '1' engages the head, must wait HEAD_LOAD_DELAYMS after this goes high before reading/writing */
        MBL = 0x40,     /* Must Be Low - ?! this must always be a '0' */
        RST = 0x80      /* software ReSeT - same as doing a hard reset to the chip */
    } WR0_BITMASKS;
#define UPD372D_WR1     REG1
    typedef enum{
        UA0W = 0x01,     /* state of pin 30 'Drv A0 Sel' line */
        UA1W = 0x02,     /* state of pin 29 'Drv A1 Sel' line */
        UAS = 0x04,     /* '1' updates pin 30/29 with UA0W/UA1W, '0' don't update pins 30/29 */
        CB3 = 0x08,     /* write clock bit strobe, change "address marks" D3:D5 with CB3:CB5 */
        CB4 = 0x10,
        CB5 = 0x20,
                CBN = 0x38,     /* clock bits for normal data */
                CBI = 0x10,     /* clock bits for index address mark */
                CBD = 0x00,     /* clock bits for ID data, or deleted data address mark */
        CBS = 0x80      /* '1' updates "address marks" with CB3:CB5, '0' don't update "address marks" */
    } WR1_BITMASKS;
#define UPD372D_WR2     REG2
    typedef enum{       /* next data byte to write to selected disk sector */
        WD0 = 0x01,
        WD1 = 0x02,
        WD2 = 0x04,
        WD3 = 0x08,
        WD4 = 0x10,
        WD5 = 0x20,
        WD6 = 0x40,
        WD7 = 0x80,
    } WR2_BITMASKS;
```

```
#define UPD372D_WR3     REG3
    typedef enum{
        CCW = 0x01,     /* set to '1' when reading/writing the last data byte, set to '0' when reading/writing the second CRC byte */
        CCG = 0x02,     /* set to '1' before reading/writing first data byte, set to '0' when writing the "address mark" */
        WER = 0x04,     /* set to '1' to disable writing to the disk */
        IXS = 0x08,     /* set to '1' (when STT=1) to start the formatting routine at the physical index hole */
        WES = 0x10,     /* set to '1' to enable writing to the disk */
        STT = 0x20,     /* Start read/write/format operation when set to '1', is automatically set to '0' when operation is complete */
        WCS = 0x40,     /* use write clock (pin 13) */
        RCS = 0x80      /* use read clock (pin 10) */
    } WR3_BITMASKS;
#define UPD372D_WR4     REG4
    typedef enum{
        UB0W = 0x01,     /* state of pin 28 'Drv B0 Sel' line */
        UB1W = 0x02,     /* state of pin 27 'Drv B1 Sel' line */
        UBS = 0x04,     /* '1' updates pin 28/27 with UB0W/UB1W, '0' don't update pins 28/27 */
        SOS = 0x20,     /* state of pin 26 'Step Out/Step' line */
        SID = 0x40,     /* state of pin 25 'Step In/Dir' line */
        STS = 0x80      /* '1' updates pin 26/25 with SOS/SID, '0' don't update pins 26/25 */
    } WR4_BITMASKS;

// There is no Write Register 5!

#define UPD372D_WR6     REG6
    typedef enum{
        DRR = 0x01,     /* '1' resets the DRQ bit (in RR0) */
        IRR = 0x02,     /* '1' resets the IRQ bit (in RR0) */
        TRR = 0x04      /* '1' resets the TRQ bit (in RR0) */
    } WR6_BITMASKS;


/* This port is not actually part of the UPD372D chip.
 *
 * It is used to briefly (defined by RC constant on U15 pins 6&7)
 * enable the FDC_ACCESS signal onto CN3 pin 36.
 *
 * This is necessary to be able to detect uPD327D interrupts as
 * they do not automatically enable this signal (as port R/W does)
 *
 * This is likely a scheme to allow support for multiple floppy controller
 * cards to run in parallel - which will never happen now.  It appears that
 * you could bypass the need for this by simply permanently enabling U33.4,
 * but for authenticity this feature is implemented in the logic.
 */
#define FDC_ACCESS_WR   REG7


#define UPD372D_RR0     REG0
    typedef enum{
        DRQ = 0x01,     /* '1' time to read/write the next data byte */
        IRQ = 0x02,     /* '1' leading edge of index hole just detected -> causes interrupt */
        TRQ = 0x04,     /* '1' 512 write clocks have occurred (~2.048mS) - can be used for timing */
        ERR = 0x08,     /* '1' one of WFT, RYA, or COR errors have occurred */
        UB0R = 0x10,     /* UB0/UB1 are the read values of the corresponding bits in WR4 */
        UB1R = 0x20,
        RYB = 0x40,     /* Drive B ready */
        ALH = 0x80      /* '1' as long as the chip has power */
    } RR0_BITMASKS;
#define UPD372D_RR1     REG1
    typedef enum{
        UA0R = 0x01,     /* UA0/UA1 are the read values of the corresponding bits in WR1 */
        UA1R = 0x02,
        WFT = 0x04,     /* Write Fault occurred */
        RYA = 0x08,     /* Drive A ready */
        COR = 0x10,     /* Command Overrun - host took too long to service DRQ */
        DER = 0x20,     /* Data Error - CRC occurred during read operation */
        T00 = 0x40,     /* '1' when drive stepper motor is positioned to track zero */
        WRT = 0x80      /* which clock read/write is being used, same as state of pin 15 */
    } RR1_BITMASKS;
#define UPD372D_RR2     REG2
    typedef enum{       /* next data byte read from the selected disk sector */
        RD0 = 0x01,
        RD1 = 0x02,
        RD2 = 0x04,
        RD3 = 0x08,
        RD4 = 0x10,
        RD5 = 0x20,
        RD6 = 0x40,
        RD7 = 0x80
```

```
    } RR2_BITMASKS;


extern UCHAR WR0_LastValue;


extern void uPD372D_Init(void);
extern void Briefly_Enable_FDC_ACCESS(void);
extern void uPD372D_WriteReg(FDC_REG_ADDR_TYPE reg, UCHAR value);
extern UCHAR uPD372D_ReadReg(FDC_REG_ADDR_TYPE reg);
extern inline void Wait_For_Interrupt(void);
extern bool uPD372D_Functional(void);

#ifdef __cplusplus
}
#endif

#endif/* UPD372D_H */
```

```
/*
 * File:   FloppyProt.h
 * Author: Alec
 *
 * Created on November 23, 2021, 7:32 AM
 */

#ifndef FLOPPYPROT_H
#defineFLOPPYPROT_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Utility.h"
#include "FDC_Interface.h"

#define FLOPPY_SECTOR_SIZE      128
#define MAX_NUM_FLOPPY_DRIVES   4

#define FLOPPY_CMD_FORMAT   0x01        /* format drive */
#define FLOPPY_CMD_READ     0x02        /* read sector */
#define FLOPPY_CMD_WRITE    0x03        /* write sector */
#define FLOPPY_CMD_STATUS   0x04        /* get status */
#define FLOPPY_CMD_IDLE     0x05        /* idle drives, host is done accessing them for the near future
                                         * (can be used by floppy controller to turn off drive motors, etc)
                                         */

// status codes
#define STATUS_DRIVE_SELECT_FAIL    0x02


typedef struct {
    UCHAR               command;
} FloppyHeaderStruct;

// FLOPPY_CMD_FORMAT
typedef struct {
    FloppyHeaderStruct  header;
    UCHAR               drive;      /* 0='A' drive, 1='B' drive, etc */
    UINT16              crc;
} FloppyFormatStruct, *pFloppyFormatStruct;

    // FLOPPY_CMD_FORMAT - REPLY
    typedef struct {
        UCHAR               status;
            #define FP_STATUS_OK        0
            #define FP_STATUS_ERROR     1
        UINT16              crc;
    }FloppyFormatReplyStruct, *pFloppyFormatReplyStruct;

// FLOPPY_CMD_READ
typedef struct {
    FloppyHeaderStruct  header;
    UCHAR               drive;      /* 0='A' drive, 1='B' drive, etc */
    TRACK_TYPE          track;      /* track to read */
    SECTOR_TYPE         sector;     /* sector to read */
    UINT16              crc;
```

```
} FloppyReadStruct, *pFloppyReadStruct;

    // FLOPPY_CMD_READ - REPLY
    typedef struct {
        UCHAR               status;     /* 0=OK, !0=error */
        UCHAR               buffer[FLOPPY_SECTOR_SIZE];
        UINT16              crc;
    }FloppyReadReplyStruct, *pFloppyReadReplyStruct;

// FLOPPY_CMD_WRITE
typedef struct {
    FloppyHeaderStruct  header;
    UCHAR               drive;      /* 0='A' drive, 1='B' drive, etc */
    TRACK_TYPE          track;      /* track to write */
    SECTOR_TYPE         sector;     /* sector to write */
    UCHAR               buffer[FLOPPY_SECTOR_SIZE];
    UINT16              crc;
} FloppyWriteStruct, *pFloppyWriteStruct;

    // FLOPPY_CMD_WRITE - REPLY
    typedef struct {
        UCHAR               status;     /* 0=OK, !0=error */
        UINT16              crc;
    }FloppyWriteReplyStruct, *pFloppyWriteReplyStruct;


// FLOPPY_CMD_STATUS
typedef struct {
    FloppyHeaderStruct  header;
    UINT16              crc;
} FloppyStatusStruct, *pFloppyStatusStruct;

    // FLOPPY_CMD_STATUS - REPLY
    typedef struct {
        FloppyHeaderStruct  header;
        UCHAR               controllerStatus;
            #define FCTRLBIT_CONNECTED      0x01            /* floppy controller is talking with controller */
        UCHAR               driveStatus[MAX_NUM_FLOPPY_DRIVES];
            #define FDRIVEBIT_CONNECTED     0x01            /* physical drive is connected to controller */
            #define FDRIVEBIT_READY         0x02            /* drive is ready to be accessed */
            #define FDRIVEBIT_DOORCLOSED    0x04            /* drive door is closed */
        UINT16              crc;
    } FloppyStatusReplyStruct, *pFloppyStatusReplyStruct;

// FLOPPY_CMD_IDLE
typedef struct {
    FloppyHeaderStruct  header;
    UINT16              crc;
} FloppyIdleStruct, *pFloppyIdleStruct;

    // FLOPPY_CMD_WRITE - REPLY
    typedef struct {
        UCHAR               status;     /* 0=OK, !0=error */
        UINT16              crc;
    }FloppyIdleReplyStruct, *pFloppyIdleReplyStruct;

#ifdef __cplusplus
}
#endif
```

```
#endif/* FLOPPYPROT_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:   SerialIF.h
 * Author: Alec
 *
 * Created on November 21, 2021, 11:04 AM
 */

#ifndef SERIALIF_H
#defineSERIALIF_H

#ifdef__cplusplus
extern "C" {
#endif

#include "Utility.h"

extern void SerialIF_Init(void);
extern void Process_Host_Commands(void);
extern void EnableSerial(bool enabled);

#ifdef__cplusplus
}
```

```
#endif

#endif/* SERIALIF_H */
```

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

#ifndef TIMER_H
#defineTIMER_H

#ifdef__cplusplus
extern "C" {
#endif

#define SUPPORT_TIMER0      0       /* Interrupt handler every 'n' mS */
#define SUPPORT_TIMER1      1       /* Free running precision timer */
#define SUPPORT_TIMER2      0       /* Free running mS timer */

// Timer Initialization
void Timer_Init(void);

#if SUPPORT_TIMER0
// 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)
#endif

#if SUPPORT_TIMER1
typedef unsigned short int PRECISIONTIMER_TYPE;
    // once the timer reaches this value it will roll over and be meaningless
    #define MAX_PRECISION_TIMER_VALUE   0xffff

    // safe upper limit for timer
    #define MAX_PRECISION_TIMER_SAFE_VALUE  (MAX_PRECISION_TIMER_VALUE - 5000)

// read precision timer value
inline PRECISIONTIMER_TYPE PrecisionTimerRead(void);
// calculate the time delta from the previous PrecisionTimerRead() value
inline PRECISIONTIMER_TYPE PrecisionTimeruSDelta(PRECISIONTIMER_TYPE prevRead);

// get the precision timer uS period
inline const PRECISIONTIMER_TYPE PrecisionTimeruSPeriod(void);
// get the max time period you can measure with the Precision Timer
// due to hardware and software constraints
inline const PRECISIONTIMER_TYPE PrecisionTimerMaxuSLimit(void);
#endif
```

```
#if SUPPORT_TIMER2
typedef unsigned char MSTIMER_TYPE;

inline MSTIMER_TYPE mSTimerRead(void);
inline const MSTIMER_TYPE mSTimerPeriod(void);
inline UINT16 mSTimerDelta(MSTIMER_TYPE prevRead);
#endif

#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, *PUCHAR;
typedef unsigned short int UINT16, *PUINT16;
typedef unsigned long int UINT32, *PUINT32;

#ifdef__cplusplus
}
#endif

#endif/* UTILITY_H */
```

```
/*
 * File:   FDC_Interface.c
 * Author: Alec
 *
 * Created on October 7, 2021, 6:08 PM
 *
 * This file has the high level public routines necessary to interface with the
 * floppy drive without any knowledge of the hardware.  The OS should use only
 * these routines to access the floppy drive.
 *
 */


/* Enable this to support hardware supported CRC generation and checking.
 *
 * Note that this adds extra processing time to the already tight logic used
 * to read/write sectors.  Using a 16F15244 PIC running at 32MHz with interrupts
 * disabled runs this logic reliably - however the timing is tight and any
 * additional code added here could break the reliability.
 *
 * If there is need to add more code, this could be disabled in the future.
 */
#define SUPPORT_READWRITE_CRC   1

#include <xc.h>
#include <pic16f15244.h>

#include "Common.h"
#include "FDC_Interface.h"
#include "FDC_Physical.h"

/* Detect_Drives
 *
 * Detect which drives are connected and working, initialize them for use.
 *
 * NOTE: using Drive_Select will only detect working drives WITH disk in AND door closed
 */
void Detect_Drives(void){

    for (int drive=0; drive<NUM_DRIVES_SUPPORTED; drive++){

        if (Drive[drive].track == NO_TRACK_SELECTED){

            if (Drive_Select((DRIVE_SELECTED_TYPE) drive)){

                Load_Head(false);

                if (FindTrackZero()){
                    Drive[drive].track = FIRST_TRACK_INDEX;
                    Drive[drive].present = true;
                }
            }
            Drive_Select(NoDriveSelected);
        }
    }
}

#define ENABLE_MOTOR_TIMEOUT    0

/* Drive Motor At Full Speed
```

```
/* Drive_Motor_At_Full_Speed
 *
 * Determine if the drive motor has reached full speed
 */
bool Drive_Motor_At_Full_Speed(void){

    bool success = true;
    PRECISIONTIMER_TYPE timer = PrecisionTimerRead();

// TODO - Figure out why this isn't working properly and uncomment the following line
//        this is not a rush as the floppy interface is working fine without this
//        so it may end up being easier simply to delete this logic.
    return success;

    // Reset any pending IRQ state
    uPD372D_WriteReg(UPD372D_WR6, IRR);

    // Set STT and RCS to enable the Index Requests (IRQ)
    uPD372D_WriteReg(UPD372D_WR3, 0x00);
    uPD372D_WriteReg(UPD372D_WR3, STT | RCS);

    // wait for the index hole to pass the sensor
    while (!(uPD372D_ReadReg(UPD372D_RR0) & IRQ) && success){


#if ENABLE_MOTOR_TIMEOUT
        // precision timer rolls over at 65535
        // abort if taking longer than this to find an index hole
        if (PrecisionTimeruSDelta(timer) > MAX_PRECISION_TIMER_SAFE_VALUE){
            success = false;
        }
#endif
    }

    // physical index hole occurred, start the timer
    timer = PrecisionTimerRead();

// IRQ is set on the leading edge of the index hole - NOT on it being under the sensor
#if 0
    // wait for this index hole to clear the sensor
    while ((uPD372D_ReadReg(UPD372D_RR0) & IRQ) && success){

#if ENABLE_MOTOR_TIMEOUT
        // precision timer rolls over at 65535
        // abort if taking longer than this to find an index hole
        if (PrecisionTimeruSDelta(timer) > MAX_PRECISION_TIMER_SAFE_VALUE){
            success = false;
        }
#endif
    }
#else
    // Reset the IRQ state
    uPD372D_WriteReg(UPD372D_WR6, IRR);
#endif

    // wait for the next index hole to pass the sensor
    while (!(uPD372D_ReadReg(UPD372D_RR0) & IRQ) && success){

#if ENABLE_MOTOR_TIMEOUT
        // precision timer rolls over at 65535
        // abort if taking longer than this to find an index hole
```

```
        if (PrecisionTimeruSDelta(timer) > MAX_PRECISION_TIMER_SAFE_VALUE){
            success = false;
        }
#endif

    }

    // Reset the IRQ state
    uPD372D_WriteReg(UPD372D_WR6, IRR);

    // if two successive index hole were timed under the MAX_PRECISION_TIMER_SAFE_VALUE
    if (success){

        // get the #of uS between the two index holes being detected
        PRECISIONTIMER_TYPE delta = PrecisionTimeruSDelta(timer);

        // determine if drive RPM is within 5 RPM of what it should be
        if (delta <= (1000000 / (DRIVE_RPM - 5))){
            success = true;
        }else{
            success = false;
        }
    }

    // Clear STT and RCS to disable the Index Requests (IRQ)
    uPD372D_WriteReg(UPD372D_WR3, 0x00);

    return success;
}



/* Select the drive to use AND wait for it to become ready
 *
 * The retries appear to only be necessary when accessing the drive immediately
 * after power up (of the drive) although there may be other times when this
 * is also true.
 *
 * The logic can't wait forever, because ready will be 'false' when the drive
 * door is open, in which case the program can't hang.
 *
 * IMPORTANT - 10000 tries may not be a good number, I just kept multiplying it
 *             by 10 until the program started working.  This could use some
 *             fine tuning.
 */
bool Drive_Select(DRIVE_SELECTED_TYPE drive){

    bool success = false;

    // wait until the drive is ready - abort when a reasonable number of attempts have failed
    for (int try=0; (try < 10000) && !success; try++){
        success = DriveSelect(drive);

        if (!success){
            // wait about 50uS
            _delay(1000);
        }else{
            // wait for the drive motor to reach full speed
            // NOT SURE IF THIS IS NEEDED - it may be that 'drive ready' also indicates that
            //   the RPM requirement has already been met.  It shouldn't hurt to leave this
```

```
            //   as a double check.
            while (!Drive_Motor_At_Full_Speed()){
                // wait about 50uS
                _delay(1000);
            }
        }
    }

    return success;
}

/* Read_Sector
 *
 * Read the specified sector
 *
 * TODO - implement retries and use NUM_READ_RETRIES define
 *
 * IMPORTANT - this is VERY speed critical
 */
READ_WRITE_STATUS_TYPE Read_Sector(TRACK_TYPE track, SECTOR_TYPE sector, SECTOR_BUFFER_TYPE buffer){

    READ_WRITE_STATUS_TYPE status = STATUS_CANT_FIND_SECTOR;

    di();

    if (Locate_Sector(track, sector)){

        UCHAR dataRead;

        status = STATUS_CANT_FIND_DATA_MARK;

        // Use write clock
        uPD372D_WriteReg(UPD372D_WR3, STT | WCS);

        // skip over the first 12 gap bytes
        for (UCHAR skip=0; skip<(POST_ID_RECORD_GAP_START_BYTES + 1); skip++){
            Wait_For_Interrupt();
        }

        /* Wait for gap byte 13.  The head has now passed the data area in the gap
         * that contains unknown information generated when the write current was
         * turned on to write the data record.
         */
        Wait_For_Interrupt();

        // reset STT, and issue read command which won't interrupt until next Data Address mark
        uPD372D_WriteReg(UPD372D_WR3, 0x00);
        uPD372D_WriteReg(UPD372D_WR3, STT | RCS);

        // get the next mark byte
        // CRC calculation begins automatically at the address mark in the read mode
        Wait_For_Interrupt();
        dataRead = uPD372D_ReadReg(UPD372D_RR2);

        if (dataRead == DATA_MARK_DATA_VALUE){

            UCHAR RR1_Status;

            // read all but the last byte of the user data into the buffer
            for (UCHAR idx=0; idx<(NUM_BYTES_PER_SECTOR - 1); idx++){
```

```
            for (uint16_t idx = 0; idx<(NUM_BYTES_PER_SECTOR - 1); idx++){
                Wait_For_Interrupt();
                buffer[idx] = uPD372D_ReadReg(UPD372D_RR2);
            }

#if SUPPORT_READWRITE_CRC
            // CCW must be set to a one while the floppy disk drive
            // head is reading the last data byte of an ID or data record
            uPD372D_WriteReg(UPD372D_WR3, STT | CCW);
#endif

            // read the last user data byte
            Wait_For_Interrupt();
            buffer[NUM_BYTES_PER_SECTOR - 1] = uPD372D_ReadReg(UPD372D_RR2);


#if SUPPORT_READWRITE_CRC
            Wait_For_Interrupt();   // 1st byte of CRC

            // CCW must be reset to a zero while the head is reading the second CRC byte.
            uPD372D_WriteReg(UPD372D_WR3, STT);

            Wait_For_Interrupt();   // 2nd byte of CRC
#endif

            RR1_Status = uPD372D_ReadReg(UPD372D_RR1);

            // Reset STT
            uPD372D_WriteReg(UPD372D_WR3, 0x00);

#if SUPPORT_READWRITE_CRC
            if (RR1_Status & DER){
                status = STATUS_INVALID_CRC;
            }else{
                status = STATUS_SUCCESS;
            }
#else
            status = STATUS_SUCCESS;
#endif

        }else{
            // reset STT
            uPD372D_WriteReg(UPD372D_WR3, 0x00);

            if (dataRead == DELETED_DATA_MARK_DATA_VALUE){
                status = STATUS_DELETED_DATA_MARK;
            }else{
                status = STATUS_INVALID_DATA_MARK;
            }
        }

    }

    ei();

    return status;
}

/* Write_Sector
 *
 * Write the specified sector
```

```
 *
 * IMPORTANT - this is VERY speed critical
 */
READ_WRITE_STATUS_TYPE Write_Sector(TRACK_TYPE track, SECTOR_TYPE sector, SECTOR_BUFFER_TYPE buffer){

    READ_WRITE_STATUS_TYPE status = STATUS_CANT_FIND_SECTOR;
    UCHAR count = NUM_BYTES_PER_SECTOR - 1;
    UCHAR *ptr = &buffer[1];

    di();

    if (Locate_Sector(track, sector)){

        // skip over the first 10 gap bytes
        for (UCHAR skip=0; skip<POST_ID_RECORD_GAP_START_BYTES-1; skip++){
            Wait_For_Interrupt();
        }

        // Set write clock logic to use 'normal' clock bits
        uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
        // Set data register to write zeroes
        uPD372D_WriteReg(UPD372D_WR2, 0x00);

        Wait_For_Interrupt();   // head is reading gap byte 11

        // Set write current and write clock to start at next BRP
        uPD372D_WriteReg(UPD372D_WR3, WCS | STT | WES);

        // if write fault occurred
        if (uPD372D_ReadReg(UPD372D_RR1) & WFT){
            // Clear the write fault flag

            WR0_LastValue |= WFR;
            uPD372D_WriteReg(UPD372D_WR0, WR0_LastValue);
            WR0_LastValue &= ~WFR;
            uPD372D_WriteReg(UPD372D_WR0, WR0_LastValue);

            status = STATUS_WRITE_FAULT;
        }else{

            Wait_For_Interrupt();   // head begins writing, write 0x00 in gap byte 12
            Wait_For_Interrupt();   // head writes 0x00 in gap byte 13

            Wait_For_Interrupt();   // head starts byte 14
            Wait_For_Interrupt();   // head starts byte 15
            Wait_For_Interrupt();   // head starts byte 16

            // Set data mark clock, and data value
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBD);
            uPD372D_WriteReg(UPD372D_WR2, DATA_MARK_DATA_VALUE);
#if SUPPORT_READWRITE_CRC
            // Set CCG so CRC calculation starts at next BRP
            uPD372D_WriteReg(UPD372D_WR3, STT | CCG);
#endif

            Wait_For_Interrupt();   // head starts data mark byte

            // Set normal clock
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
```

```
#if SUPPORT_READWRITE_CRC
            // Reset CCG (CCG must be reset before next BRP or CRC calculation would begin again)
            uPD372D_WriteReg(UPD372D_WR3, STT);
#endif

            /* The first user data byte write is being separated from the others
             * due to timing constraints.  It takes too long to update WR2 when
             * all of the user bytes are written in the 'for' loop.
             */

            // write out the first user data byte
            uPD372D_WriteReg(UPD372D_WR2, buffer[0]);
            Wait_For_Interrupt();


            /* Now write the remainder of the user data bytes.  This can be
             * done in the 'for' loop now because there is more time now that
             * WR1 and WR3 don't need to be updated between bytes.
             */

            // write all of the other user data in the sector
#if 0
            for (UCHAR idx=1; idx<NUM_BYTES_PER_SECTOR; idx++){
                uPD372D_WriteReg(UPD372D_WR2, buffer[idx]);
                Wait_For_Interrupt();
            }
#else
            // this is more obtuse, BUT runs fast enough to keep up with the data stream
            do{
                uPD372D_WriteReg(UPD372D_WR2, *ptr++);
                Wait_For_Interrupt();
            }while (--count != 0);
#endif

            // head is writing the last of the user data bytes now

#if SUPPORT_READWRITE_CRC
            // Set CCW.  In write mode the chip will begin writing bits from
            // the CRC register at the next BRP following the setting of CCW.
            uPD372D_WriteReg(UPD372D_WR3, STT | CCW);

            Wait_For_Interrupt();   // head starts writing first CRC byte
            Wait_For_Interrupt();   // head starts writing second CRC byte

            // Reset CCW.  CRC bit writing will stop at next BRP
            uPD372D_WriteReg(UPD372D_WR3, STT);
#endif

            uPD372D_WriteReg(UPD372D_WR2, GAP_START_DATA_VALUE);

            Wait_For_Interrupt();   // head is writing gap byte

            // Set write enable reset.  Write current will stop at next BRP
            uPD372D_WriteReg(UPD372D_WR3, STT | WER);
            Wait_For_Interrupt();   // head begins writing 2nd gap byte


            // Reset STT
            uPD372D_WriteReg(UPD372D_WR3, 0x00);

            status = STATUS_SUCCESS;
```

```
        }
    }

    ei();

    return status;
}


/* Format_Drive
 *
 * This routine will format the media in the selected drive
 *
 */
bool Format_Drive(void){

    bool success = false;

    if (Drive_Selected != NoDriveSelected){

        FindTrackZero();
        Load_Head(true);

        for (UCHAR track=FIRST_TRACK_INDEX; track<(NUM_TRACKS_PER_DRIVE + FIRST_TRACK_INDEX); track++){

            // Setup to write post index hole gap data using clock bits for normal data
            //
            // IMPORTANT - the post physical index hole logic is different
            // than all the other gaps, in that the clock/data values must be
            // setup BEFORE STT is set because the FCC will use these values as
            // soon as the physical index hole is detected.
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
            uPD372D_WriteReg(UPD372D_WR2, GAP_START_DATA_VALUE);

            // Reset STT, then setup to start writing at index hole
            uPD372D_WriteReg(UPD372D_WR3, 0x00);
            uPD372D_WriteReg(UPD372D_WR3, WCS | STT | WES | IXS);

            // Wait for index hole found, then reset index request
            Wait_For_Interrupt();       // head is writing first gap byte as soon as index is found
            uPD372D_WriteReg(UPD372D_WR6, IRR);

            // issue the post index hole gap
            Write_Gap(POST_INDEX_HOLE_GAP_START_BYTES - 1, POST_INDEX_HOLE_GAP_END_BYTES, ClockNormal);

            //----------------------------------------
            // write index mark
            //----------------------------------------

            // Setup to write index mark using clock bits for index marks
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBI);
            uPD372D_WriteReg(UPD372D_WR2, INDEX_ADDRESS_MARK_DATA_VALUE);
            Wait_For_Interrupt();

            // issue the post index mark gap
            Write_Gap(POST_INDEX_MARK_GAP_START_BYTES - 1, POST_INDEX_MARK_GAP_END_BYTES, ClockNormal);

            // write all of the sectors on this track
            for (UCHAR sector=FIRST_SECTOR_INDEX; sector<(NUM_SECTORS_PER_TRACK + FIRST_SECTOR_INDEX); sector++){
```

```
                //---------------------------------------
                // write ID record
                //---------------------------------------

// this code was commented out and may have been causing all sectors (after the first) to fail
// re-enable this code and see if it works better.
// ===>  Yes this code IS REQUIRED!  NOT sure why I removed it, leave here for a while just in case.
#define ENABLE_COMMENTEDOUT_CODE    1

                // Setup to write data using clock bits for ID data
                uPD372D_WriteReg(UPD372D_WR1, CBS | CBD);
                uPD372D_WriteReg(UPD372D_WR2, ID_ADDRESS_MARK_DATA_VALUE);
                // Set CCG, this causes CRC calculation to begin at next BRP
#if ENABLE_COMMENTEDOUT_CODE
                uPD372D_WriteReg(UPD372D_WR3, STT | CCG);
#endif


                Wait_For_Interrupt();

                // Setup to write data using clock bits for normal data
                uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
                // Reset CCG, this must be done before next BRP or CRC calculation would begin again
#if ENABLE_COMMENTEDOUT_CODE
                uPD372D_WriteReg(UPD372D_WR3, STT);
#endif
                uPD372D_WriteReg(UPD372D_WR2, track);

                Wait_For_Interrupt();

                uPD372D_WriteReg(UPD372D_WR2, 0x00);

                Wait_For_Interrupt();

                uPD372D_WriteReg(UPD372D_WR2, sector);

                Wait_For_Interrupt();

                uPD372D_WriteReg(UPD372D_WR2, 0x00);

                Wait_For_Interrupt();

                // Set CCW, in write mode the chip will begin writing bits from the CRC
                // registers at the next BRP following setting of CCW.
                uPD372D_WriteReg(UPD372D_WR3, STT | CCW);

                Wait_For_Interrupt();       // CRC is two bytes long
                Wait_For_Interrupt();

                // issue the post ID record gap
                Write_Gap(POST_ID_RECORD_GAP_START_BYTES, POST_ID_RECORD_GAP_END_BYTES, ClockNormal);

                //---------------------------------------
                // write Data Field record
                //---------------------------------------

                // Setup to write data address mark using clock bits for ID data
                uPD372D_WriteReg(UPD372D_WR1, CBS | CBD);
                uPD372D_WriteReg(UPD372D_WR2, DATA_MARK_DATA_VALUE);

                // Set CCG, this causes CRC calculation to begin at next BRP
```

```
                uPD372D_WriteReg(UPD372D_WR3, STT | CCG);

                Wait_For_Interrupt();

                // Setup to write user data using clock bits for normal data
                uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
                // Reset CCG, this must be done before next BRP or CRC calculation would begin again
                uPD372D_WriteReg(UPD372D_WR3, STT);
                uPD372D_WriteReg(UPD372D_WR2, FORMATTED_DATA_VALUE);

                // write all of the user data bytes with the same FORMATTED_DATA_VALUE
                for (int count=0; count<NUM_BYTES_PER_SECTOR; count++){
                    Wait_For_Interrupt();
                }

                // Set CCW, in write mode the chip will begin writing bits from the CRC
                // registers at the next BRP following setting of CCW.
                uPD372D_WriteReg(UPD372D_WR3, STT | CCW);

                Wait_For_Interrupt();       // CRC is two bytes long
                Wait_For_Interrupt();

                // issue the data record gap
                Write_Gap(POST_DATA_RECORD_GAP_START_BYTES, POST_DATA_RECORD_GAP_END_BYTES, ClockNormal);
            }

            //----------------------------------------
            // write 0xff until the end of the track
            //----------------------------------------

            uPD372D_WriteReg(UPD372D_WR2, 0xff);

            // wait until physical index hole detected
            // STT must be active for IRQ to register
            uPD372D_WriteReg(UPD372D_WR3, STT);
            Wait_For_Interrupt();   // index interrupt (will be cleared below when STT is cleared)

            //----------------------------------------
            // disable write and move head to next track
            //----------------------------------------


            // Write enable and stt reset (index request is automatically reset by stt reset)
            uPD372D_WriteReg(UPD372D_WR3, WER);

            // Wait for tunnel erase head to reach end of track
            DelayMs(2);

            // Step head in one track
            StepHeadIn(true);
        }

        success = true;
    }

    return success;
}

#if ENABLE_CHECKDISK_LOGIC
#define WRITE_CRC               0
```

```
#define CHECKDISK_TRACK_BYTES   128
UINT16 goodBytes[NUM_TRACKS_PER_DRIVE];

/* CheckDisk
 *
 * This routine will destructively verify the media on the specified drive
 *
 * This has been verified to work properly 10/27/2021 (with WRITE_CRC = 0)
 *
 * This was originally written as a means to debug the drive read/write logic
 * without all of the overly complicated format requirements.  This is not
 * necessary for normal operation and is only left here as a means to debug
 * complicated situations that may arise in the future.  Eventually this
 * could be removed.
 */
bool CheckDisk(UCHAR drive){

    bool success = Drive_Select(drive);
    UCHAR goodTracks;
    UCHAR byteRead;


    if (success){

        FindTrackZero();
        Load_Head(true);

        Drive[Drive_Selected].track = FIRST_TRACK_INDEX;

        for (UCHAR track=FIRST_TRACK_INDEX; track<NUM_TRACKS_PER_DRIVE; track++){

            // Setup to write post index hole gap data using clock bits for normal data
            //
            // IMPORTANT - the post physical index hole logic is different
            // than all the other gaps, in that the clock/data values must be
            // setup BEFORE STT is set because the FCC will use these values as
            // soon as the physical index hole is detected.


            // Setup to write 2x GAP_START_DATA_VALUE gap bytes
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
            uPD372D_WriteReg(UPD372D_WR2, GAP_START_DATA_VALUE);

            // Reset STT, then setup to start writing at index hole
            uPD372D_WriteReg(UPD372D_WR3, 0x00);
            uPD372D_WriteReg(UPD372D_WR3, WCS | STT | WES | IXS);

            // Wait for index hole found, then reset index request
            Wait_For_Interrupt();       // head is writing 1st start gap byte as soon as index is found
            uPD372D_WriteReg(UPD372D_WR6, IRR);

            Wait_For_Interrupt();       // head is writing 2nd start gap byte

            // Setup to write 2x GAP_END_DATA_VALUE gap bytes
            uPD372D_WriteReg(UPD372D_WR2, GAP_END_DATA_VALUE);

            Wait_For_Interrupt();       // head is writing 1st end gap byte
            Wait_For_Interrupt();       // head is writing 2nd end gap byte

            // Setup to write address mark using clock bits for address marks
```

```
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBD);
            uPD372D_WriteReg(UPD372D_WR2, ID_ADDRESS_MARK_DATA_VALUE);


#if WRITE_CRC
            // Set CCG, this causes CRC calculation to begin at next BRP (address mark)
            uPD372D_WriteReg(UPD372D_WR3, STT | CCG);
#endif

            Wait_For_Interrupt();       // head is writing address mark byte

#if WRITE_CRC
            // Clear CCG, this is required to prevent the CRC calculation from
            // restarting on each data byte
            uPD372D_WriteReg(UPD372D_WR3, STT);
#endif

            // Setup to write the track number as the data bytes
            uPD372D_WriteReg(UPD372D_WR1, CBS | CBN);
            uPD372D_WriteReg(UPD372D_WR2, track);

            // write a bunch of data to the track
            for (int x=0; x<CHECKDISK_TRACK_BYTES; x++){
                Wait_For_Interrupt();
            }

#if WRITE_CRC
            // Set CCW, in write mode the chip will begin writing bits from the CRC
            // registers at the next BRP following setting of CCW.
            uPD372D_WriteReg(UPD372D_WR3, STT | CCW);

            Wait_For_Interrupt();       // CRC is two bytes long
            Wait_For_Interrupt();

            // clear CCW to stop writing CRC data
            uPD372D_WriteReg(UPD372D_WR3, STT);
#endif

            // write two 0x00 pad bytes after the data to make sure
            // the last data byte actually got written out
            uPD372D_WriteReg(UPD372D_WR2, 0x00);
            Wait_For_Interrupt();
            Wait_For_Interrupt();



            // clear STT to stop writing
            uPD372D_WriteReg(UPD372D_WR3, 0x00);

            // Wait for tunnel erase head to reach end of track
            DelayMs(2);


            //----------------------------------------
            // read the data back
            //----------------------------------------

            // Wait for the index hole to come back around so that we aren't
            // reading garbage from the end of the track that was left over
            // STT must be active for IRQ to register
            uPD372D_WriteReg(UPD372D_WR3, STT);
```

```
            Wait_For_Interrupt();   // index interrupt (will be cleared below when STT is cleared)


            // Reset STT, and issue read command
            // (interrupts will start after ID_ADDRESS_MARK_DATA_VALUE mark is detected)
            uPD372D_WriteReg(UPD372D_WR3, 0x00);
            uPD372D_WriteReg(UPD372D_WR3, STT | RCS);

            // The next ID address mark, Data address mark or Deleted Data
            // address mark read by the disk drive causes an interrupt
            // request and a BRP. Interrupts continue to occur as each
            // byte is read and at each physical index until STT is
            // reset.
            Wait_For_Interrupt();   // mark interrupt

            // read in the address mark
            byteRead = uPD372D_ReadReg(UPD372D_RR2);

            goodBytes[track]=0;

            if (byteRead == ID_ADDRESS_MARK_DATA_VALUE){

                // attempt to read all of the bytes in
                for (int x=0; x<CHECKDISK_TRACK_BYTES; x++){

                    Wait_For_Interrupt();


                    byteRead = uPD372D_ReadReg(UPD372D_RR2);

                    if (byteRead == track){
                        goodBytes[track]++;
                    }
                }
            }

            //---------------------------------------
            // disable read and move head to next track
            //---------------------------------------

            // Reset STT
            uPD372D_WriteReg(UPD372D_WR3, 0x00);

            // Step head in one track
            StepHeadIn(true);
        }

        Load_Head(false);

        for (int x=0, goodTracks=0; x<NUM_TRACKS_PER_DRIVE; x++){
            if (goodBytes[x] == CHECKDISK_TRACK_BYTES){
                goodTracks++;
            }
        }

        if (goodTracks == NUM_TRACKS_PER_DRIVE){
            success = true;
        }else{
            success = false;
        }
```

```
    }

    return success;
}
#endif
```

```
/*
 * File:   FDC_Logic.c
 * Author: Alec
 *
 * Created on October 7, 2021, 6:08 PM
 *
 * This file has various private utility logic necessary to support the code in
 * FDC_Interface.c.  These routines should NOT be called from anywhere outside
 * of FDC_Interface.c.
 *
 */


#include <xc.h>
#include <pic16f15244.h>

#include "Common.h"
#include "FDC_Logic.h"
#include "FDC_Interface.h"
#include "FDC_Physical.h"


// ID record preceeding each sector
// (don't use TRACK_TYPE and SECTOR_TYPE here as this structure is an industry standard and thus unchangeable)
typedef struct{
    UCHAR ID_Addres_Mark;
    UCHAR Track_Address;
    UCHAR ZeroByte1;
    UCHAR Sector_Address;
    UCHAR ZeroByte2;
    UCHAR CRC1;
    UCHAR CRC2;
} FDC_ID_RECORD;


DRIVE_SELECTED_TYPE Drive_Selected;
DRIVE_TYPE Drive[NUM_DRIVES_SUPPORTED];

void uPD372D_Logic_Init(void){

    // initialize all of the drive structures
    for (UCHAR drive=0; drive<NUM_ENTRIES(Drive); drive++){

        Drive[drive].present = false;
        Drive[drive].headLoaded = false;
        Drive[drive].readWriteWorked = false;
        Drive[drive].track = NO_TRACK_SELECTED;
    }

    // default to no drive is active
    Drive_Selected = NoDriveSelected;
    DriveSelect(Drive_Selected);
}

/* Select the drive to use
 *   drive
 *      DriveA -> DRV SEL1 activated (DRVA0 SEL on FDC)
 *      DriveB -> DRV SEL2 activated (DRVA1 SEL on FDC)
 *      DriveC -> DRV SEL3 activated (DRVB0 SEL on FDC)
 *      DriveD -> DRV SEL4 activated (DRVB1 SEL on FDC)
 *      NoDriveSelected -> deselects all drives
 *
 *  Note: signals on floppy drive are active low, however FDC board has an
 *          inverter on these signals so they should be driven active high
```

```
 *          here.
 *
 *  returns: true if successful
 */
bool DriveSelect(DRIVE_SELECTED_TYPE drive){

    bool success = true;

    if ((drive != Drive_Selected) || (drive == NoDriveSelected)){

        // if changing drives from a real drive, then unload the head on the last active drive
        if (Drive_Selected != NoDriveSelected){
            Load_Head(false);
        }

        switch (drive){
            case DriveA:
                // set the drive select bits for the 1st drive
                uPD372D_WriteReg(UPD372D_WR1, UAS | UA0W);       // set UA0 high (active)
                success = uPD372D_ReadReg(UPD372D_RR1) & RYA;
                break;
            case DriveB:
                // set the drive select bits for the 2nd drive
                uPD372D_WriteReg(UPD372D_WR1, UAS | UA1W);       // set UA1 high (active)
                success = uPD372D_ReadReg(UPD372D_RR1) & RYA;
                break;
            case DriveC:
                // set the drive select bits for the 3rd drive
                uPD372D_WriteReg(UPD372D_WR4, UBS | UB0W);      // set UB0 high (active)
                success = uPD372D_ReadReg(UPD372D_RR0) & RYB;
                break;
            case DriveD:
                // set the drive select bits for the 4th drive
                uPD372D_WriteReg(UPD372D_WR4, UBS | UB1W);      // set UB1 high (active)
                success = uPD372D_ReadReg(UPD372D_RR0) & RYB;
                break;
            default:
                success = false;
            case NoDriveSelected:
    //TODO figure out why I cant turn off both drives??
                uPD372D_WriteReg(UPD372D_WR1, UAS);             // deselect UA0 & UA1
                uPD372D_WriteReg(UPD372D_WR4, UBS);             // deselect UB0 & UB1
                break;
        }

        if (success){
            Drive_Selected = drive;
        }
    }


    return success;
}


/* Delay a specific #of mS
 */
void DelayMs(UCHAR count){

    PRECISIONTIMER_TYPE last = PrecisionTimerRead();

    while (PrecisionTimeruSDelta(last) < (count * 1000)) ;
}
```

```
/* FindTrackZero
 *
 * Move the head to track zero
 *
 * This MUST be called once prior to calling Seek
 */
bool FindTrackZero(void){

    UCHAR attempts = 0;
    bool success = false;

    // StepHeadIn won't work if track = NO_TRACK_SELECTED
    Drive[Drive_Selected].track = NO_TRACK_SELECTED - 1;

    // step head out until track 0 found
    while (!(uPD372D_ReadReg(UPD372D_RR1) & T00) && (attempts < NUM_TRACKS_PER_DRIVE)){
        StepHeadIn(false);
        attempts++;
    }

    if (uPD372D_ReadReg(UPD372D_RR1) & T00){
        Drive[Drive_Selected].track = FIRST_TRACK_INDEX;
        success = true;
    }else{
        // couldn't find track 0
        Drive[Drive_Selected].track = NO_TRACK_SELECTED;
    }

    return success;
}



/* Seek
 *
 * Move the head to the specified track
 *
 * track - track to seek
 */
bool Seek(UCHAR track){

    bool success = false;

    if (Drive[Drive_Selected].track != NO_TRACK_SELECTED){

        while (track != Drive[Drive_Selected].track){

            // step 'in' if requested track is greater than current track
            StepHeadIn(track > Drive[Drive_Selected].track);
        }

        success = true;
    }

    return success;
}


/* StepHeadIn
 *
 * Step the head one track in the specified direction
 *
```

```
 * inDirection - true if stepping 'in' (else 'out')
 */
bool StepHeadIn(bool inDirection){

    bool success = false;

    if (Drive[Drive_Selected].track != NO_TRACK_SELECTED){

        // update the track number
        Drive[Drive_Selected].track += inDirection ? +1 : -1;

        // update the write current based upon the new track number
        if (Drive[Drive_Selected].track > LAST_OUTER_TRACK){
            // use high current writes
            WR0_LastValue &= ~LCT;
        }else{
            // use low current writes
            WR0_LastValue |= LCT;
        }
        // write the new value to the chip
        uPD372D_WriteReg(UPD372D_WR0, WR0_LastValue);

        // issue the step pulse with the requested direction
        uPD372D_WriteReg(UPD372D_WR4, (inDirection ? SID : 0x00) | SOS | STS);

        // wait for stepper motor to settle
        DelayMs(STEP_DELAYMS);

        // clear the step pulse
        uPD372D_WriteReg(UPD372D_WR4, (inDirection ? SID : 0x00) | STS);

        success = true;
    }

    return success;
}



/* Load_Head
 *
 * Load/Unload the head on the selected drive
 */
void Load_Head(bool load){

    if (load){
        if (!Drive[Drive_Selected].headLoaded){

            // set head load bit
            WR0_LastValue |= HLD;

            // write the new value to the chip
            uPD372D_WriteReg(UPD372D_WR0, WR0_LastValue);

            // wait for head to settle
            DelayMs(HEAD_LOAD_DELAYMS);

            Drive[Drive_Selected].headLoaded = true;
        }
    }else{
        if (Drive[Drive_Selected].headLoaded){

            // clear head load bit
```

```
            WR0_LastValue &= ~HLD;

            // write the new value to the chip
            uPD372D_WriteReg(UPD372D_WR0, WR0_LastValue);

            Drive[Drive_Selected].headLoaded = false;
        }
    }
}

/* Write an industry standard gap sequence to the drive
 *
 * startCount - #of bytes of GAP_START_DATA_VALUE (followed by)
 * endCount - #of bytes of GAP_END_DATA_VALUE
 * clockType - type of clock pulses to use on gap bytes
 */
void Write_Gap(UCHAR startCount, UCHAR endCount, FDC_CLOCK_TYPE clockType){

    static UCHAR lastClockType = 0xff;

    if (clockType != lastClockType){
        uPD372D_WriteReg(UPD372D_WR1, CBS | clockType);
        lastClockType = clockType;
    }
    uPD372D_WriteReg(UPD372D_WR2, GAP_START_DATA_VALUE);

    // write remaining gap start bytes
    for (int x=0; x<(startCount); x++){
        Wait_For_Interrupt();
    }

    // Change to use gap end data value
    uPD372D_WriteReg(UPD372D_WR2, GAP_END_DATA_VALUE);


    // write remaining gap end bytes
    for (int x=0; x<(endCount); x++){
        Wait_For_Interrupt();
    }
}

/* Locate_Sector
 *
 * Locate the specified track/sector in preparation for a sector read/write operation
 *
 * It is assumed that the correct drive has already been selected
 *
 * This routine (if successful) will leave the FDC state machine in the read state
 * in the Gap2 area (between the Record ID and Data Field records.  This allows
 * the logic calling this routine to read/write the actual user data in the
 * Data Field area.
 *
 * IMPORTANT - this is VERY speed critical
 */
bool Locate_Sector(TRACK_TYPE track, SECTOR_TYPE sector){

    bool success = false;
    FDC_ID_RECORD idRecord;
    UCHAR byteRead;

    if ((track < (NUM_TRACKS_PER_DRIVE + FIRST_TRACK_INDEX)) && (sector < (NUM_SECTORS_PER_TRACK + FIRST_SECTOR_INDEX))){

        if (Seek(track)){

            Load_Head(true);
```

```
        for (UCHAR retry=0; retry<MAX_REVOLUTIONS_TO_FIND_SECTOR_LIMIT; ){

            // Reset STT, and issue read command
            // (interrupts will start after ID_ADDRESS_MARK_DATA_VALUE mark is detected)
            uPD372D_WriteReg(UPD372D_WR3, 0x00);
            uPD372D_WriteReg(UPD372D_WR3, STT | RCS);

            // The next ID address mark, Data address mark or Deleted Data
            // address mark read by the disk drive causes an interrupt
            // request and a BRP. Interrupts continue to occur as each
            // byte is read and at each physical index until STT is
            // reset.
            Wait_For_Interrupt();

            byteRead = uPD372D_ReadReg(UPD372D_RR2);

            if (byteRead == ID_ADDRESS_MARK_DATA_VALUE)
            {

                Wait_For_Interrupt();
                idRecord.Track_Address = uPD372D_ReadReg(UPD372D_RR2);
                Wait_For_Interrupt();
                idRecord.ZeroByte1 = uPD372D_ReadReg(UPD372D_RR2);
                Wait_For_Interrupt();
                idRecord.Sector_Address = uPD372D_ReadReg(UPD372D_RR2);

                /* Set CCW
                 *
                 * The Bit Ring Pulse (BRP) following the setting of CCW will start
                 * a bit by bit comparison of the data read from the disk with the
                 * data read from the CRC register.
                 *
                 * Although the CPU will read the completed 2nd zero byte at the
                 * next BRP, the disk drive head will begin reading the 1st CRC byte.
                 */
                uPD372D_WriteReg(UPD372D_WR3, STT | CCW);

                Wait_For_Interrupt();
                idRecord.ZeroByte2 = uPD372D_ReadReg(UPD372D_RR2);
                Wait_For_Interrupt();
                idRecord.CRC1 = uPD372D_ReadReg(UPD372D_RR2);

                // Reset CCW bit (at next BRP bit by bit CRC comparison will end)
                uPD372D_WriteReg(UPD372D_WR3, STT);

                Wait_For_Interrupt();
                idRecord.CRC2 = uPD372D_ReadReg(UPD372D_RR2);

                // this first check should always work on a properly formatted disk
                if ((idRecord.ZeroByte1 == 0x00) &&
                    (idRecord.ZeroByte2 == 0x00) &&
                    (idRecord.Track_Address == track)
                ){
                    // this will fail repeatedly until the correct sector is reached
                    if ((idRecord.Sector_Address == sector) /* &&
                        !(uPD372D_ReadReg(UPD372D_RR1) & DER) */
                    ){

                        // the desired track/sector has been found w/o any error conditions
                        success = true;

                        // break; out of retry loop
```

```
                        break;
                    }
                }else{
                    // disk is corrupted, Zero/Track bytes are invalid
                    success = false;
                }
            }

            // get here if any errors, not the correct track/sector
            if (uPD372D_ReadReg(UPD372D_RR0) & IRQ){
                // physical index hole occurred, indicate we used up another one of our retries
                retry++;
            }
        }
    }

    if (!success){
        // if for any reason, the track/sector could not be found, reset STT
        uPD372D_WriteReg(UPD372D_WR3, 0x00);

        // TODO this should probably also unload the head?? - not in example code
    }
}

return success;
}
```

```
/*
 * File:   FDC_Physical.h
 * Author: Alec
 *
 * Created on October 7, 2021, 6:08 PM
 *
 * This file has the bare minimum logic necessary to interface to the uPD372D physical chip.
 *
 */

#define TRY_TO_GET_SERIAL_WORKING           0
#define USE_ADDRESS_BITS_FIX                0
#define FORCE_PORTC_TO_INPUTS_AFTER_WRITE   1

#include <xc.h>
#include <pic16f15244.h>

#include "Common.h"
#include "FDC_Physical.h"
#include "FDC_Logic.h"

UCHAR WR0_LastValue;        // Only WR0 needs to keep track of the last values written

void uPD372D_Init(void){

    // set data bus to inputs as default
    TRISC = 0xff;

    // set FDC address lines as outputs
    TRISAbits.TRISA2 = 0;
    TRISAbits.TRISA4 = 0;
    TRISAbits.TRISA5 = 0;

    // set !IORD and !IOWR as outputs
    TRISBbits.TRISB6 = 0;
    TRISBbits.TRISB7 = 0;

    // set FDCAccess (RB5) as input
    TRISBbits.TRISB5 = 1;



    // set !IORD (RB6) to inactive
    LATBbits.LATB6 = 1;
    // set !IOWR (RB7) to inactive
    LATBbits.LATB7 = 1;

    WR0_LastValue = 0x00;

    // Reset the chip
    uPD372D_WriteReg(UPD372D_WR0, RST);

    // Turn the RST bit off in our copy of WR0 (this happens automatically in the chip)
    WR0_LastValue &= ~RST;

    // initialize the upper level driver logic
    uPD372D_Logic_Init();
```

```
}


/* IMPORTANT NOTE ABOUT "FDC ACCESS"
 *
 * The logic can't wait for FDC ACCESS to become inactive BEFORE attempting a read/write
 * This can't be done because FDCACCESS is tri-stated by the FDC controller PCB after
 * the last read/write access is completed.  Looking at this line at other times
 * results in a slowly decaying floating signal that takes dozens of mS to decay
 * low enough to read an inactive state.
 *
 * If the FDC PCB were modified to always enable U33.4 this would be allowable.
 */


/* uPD372D_WriteReg         (~750uS?)
 *
 * reg - one of UPD372D_XR9 (see uPD372D.h)
 * value - new value to write to FDC register
 *
 * IMPORTANT - this is VERY speed critical
 */
void uPD372D_WriteReg(FDC_REG_ADDR_TYPE reg, UCHAR value){

    // Setup the FDC address bits (UPD372D_BASE is the full address)
#if !TRY_TO_GET_SERIAL_WORKING && !USE_ADDRESS_BITS_FIX
    // updating the entire PortA may interfere with the debugger tool
    // which uses RA0/RA1/RA3
    LATA = reg;
#else
    LATAbits.LATA2 = reg & 0x04;
    LATAbits.LATA4 = reg & 0x10;
    LATAbits.LATA5 = reg & 0x20;
#endif

    // Setup the value to write the port
    LATC = value;

    if (reg == REG0){
        // update the last value written to WR0
        WR0_LastValue = value;
    }


#if !TRY_TO_GET_SERIAL_WORKING
    // REMOVING THIS ONE LINE MAKES THE SERIAL RECEIVE START TO WORK
    //  BUT LEAVING THIS LINE IN AND LATER SETTING THE PORT BACK TO
    //  INPUTS DOES **NOT** WORK!  THIS IS VERY BIZARRE.

    // ensure data bus is outputs
    TRISC = 0x00;
#endif

    // pulse the !IOWR line
    LATBbits.LATB7 = 0;
```

```
    /* There is about a 100nS delay from !IOWR actually going low to FDC Access
     * being enabled and read by the PIC as a high signal.  At 32MHz PIC instruction
     * cycle clock, it appears that each instruction takes about 100nS.
     *
     * So there needs to be at least this amount of delay between activating
     * !IORD and starting to wait for FDC Access going low, otherwise the
     * logic will IMMEDIATELY see FDC Access as low and terminate the write cycle
     * too soon - potentially writing garbage.
     */
    _delay(1);


    // wait for FDCAccess to go inactive
    // this stretches the access out to about 2.5uS?
    // with the FDC card removed, it appears that this line floats low so this won't hang
    while (PORTBbits.RB5 == 1) ;

    LATBbits.LATB7 = 1;

#if TRY_TO_GET_SERIAL_WORKING || FORCE_PORTC_TO_INPUTS_AFTER_WRITE
    // ensure data bus is inputs
    TRISC = 0xff;
#endif
}

/* uPD372D_ReadReg      (~1mS)
 *
 * reg - one of UPD372D_XR9 (see uPD372D.h)
 *
 * IMPORTANT - this is VERY speed critical
 */
UCHAR uPD372D_ReadReg(FDC_REG_ADDR_TYPE reg){

    UCHAR retVal;

    // Setup the FDC address bits (UPD372D_BASE is the full address)
#if !TRY_TO_GET_SERIAL_WORKING && !USE_ADDRESS_BITS_FIX
    // updating the entire PortA may interfere with the debugger tool
    // which uses RA0/RA1/RA3
    LATA = reg;
#else
    LATAbits.LATA2 = reg & 0x04;
    LATAbits.LATA4 = reg & 0x10;
    LATAbits.LATA5 = reg & 0x20;
#endif

    // activate the !IORD line
    LATBbits.LATB6 = 0;

    /* There is about a 100nS delay from !IORD actually going low to FDC Access
     * being enabled and read by the PIC as a high signal.  At 32MHz PIC instruction
     * cycle clock, it appears that each instruction takes about 100nS.
     *
     * So there needs to be at least this amount of delay between activating
     * !IORD and starting to wait for FDC Access going low, otherwise the
     * logic will IMMEDIATELY see FDC Access as low and read the data in
```

```
     * too soon - essentially reading garbage.
     */

    // ensure data bus is inputs
    // shouldn't be necessary, but since a delay is needed anyway this is more useful than _delay(1)
    TRISC = 0xff;

    // wait for FDCAccess to go inactive
    // this stretches the access out to about 2.5uS
    // with the FDC card removed, it appears that this line floats low so this won't hang
    while (PORTBbits.RB5 == 1) ;

    // Read the port value
    retVal = PORTC;

    // deactivate the !IORD line
    LATBbits.LATB6 = 1;

    return retVal;
}

/* Wait_For_Interrupt
 *
 * using this without optimization turned on adds about 1uS because of the call
 * TODO look into making this a define so it can be used efficiently during debugging
 *
 * IMPORTANT - this is VERY speed critical
 */
inline void Wait_For_Interrupt(void){

    // briefly enable the uPD372D 'FDC ACCESS' signal onto CN3 pin 36 until INT (interrupt) becomes active
    // this will drive FDC ACCESS high until interrupt, which will then go low
    // ASSUMING that you have provided a 1K pull-down resistor on the FDC ACCESS line.
    uPD372D_WriteReg(FDC_ACCESS_WR, 0x00);

    // wait for FDCAccess to go high
    //while (PORTBbits.RB5 == 0) ;


    // wait for FDCAccess to go low
    while (PORTBbits.RB5 == 1) ;
}


/* uPD372D_Functional
 *
 * Determine if the uPD372D chip is present and functional
 */
bool uPD372D_Functional(void){

    bool functional = false;

    if (uPD372D_ReadReg(UPD372D_RR0) & ALH){

        // Always High bit detected - presumably the FDC chip is functioning
        functional = true;
    }
```

```
    return functional;
}
```

```
/*
 * 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 =   OFF
#pragma config VDDAR =      HI
#pragma config MCLRE =      EXTMCLR
#pragma config PWRTS =      PWRT_OFF
#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 "Common.h"
#include "Timer.h"
#include "FDC_Physical.h"
#include "FDC_Interface.h"
#include "SerialIF.h"
#include "RemapPins.h"



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

    // invoke other interrupt modules
#if SUPPORT_TIMER0
    Timer0_Handler();
#endif
}

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

    // Set all bits to digital (not analog) functions)
    ANSELA = 0x00;
    ANSELB = 0x00
```

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

    TRISC = 0xff;

    /* IMPORTANT
     *
     * PortC is shared between the data bus for the FDC board, AND the TX/RX
     * lines for the serial port to the host processor.  On power-up the PIC
     * port lines default to inputs so there will be no conflict between
     * the shared lines, even if the serial lines are gated onto the PortC
     * lines.
     *
     * If however, PortC is configured as an output port BEFORE the serial
     * drivers are disabled, the two output drivers will conflict, potentially
     * sending one, or both, of the drivers into latch up thereby preventing
     * the serial port from receiving characters until the latch up condition
     * is resolved by power cycling the PIC.
     *
     * Therefore, it is critical to disable the serial port latch immediately
     * upon power before any other operation is attempted.
     */
    // Configure RB4 (which enables U2 onto the PortC bus) as an output that is high
    TRISBbits.TRISB4 = 0;
    LATBbits.LATB4 = 1;


    Timer_Init();
    uPD372D_Init();

    // SerialIF_Init MUST be after uPD372D_Init as they both try to initialize the shared PORTC configuration
    SerialIF_Init();

    EnableSerial(false);
    Detect_Drives();
    EnableSerial(true);


    ei(); // enable all interrupts

    while (1){

        /* It is not safe to do any operations on the floppy drive in the main loop
         * because that disconnects the serial port from the host communication
         * and will cause host transmitted characters to be dropped while
         * accessing the floppy drive.
         *
         * It is preferable to move this logic to the host side by providing
         * a host command that allows it to perform the necessary functionality.
         */

        // process commands received over the host serial port
        Process_Host_Commands();
    }
}
```

```
#include <xc.h>
#include <pic16f15244.h>

#include "Common.h"
#include "RemapPins.h"
#include "SerialIF.h"
#include "FloppyProt.h"
#include "FDC_Interface.h"
#include "Timer.h"

extern void EnableSerial(bool enabled);
extern bool SerialByteAvailable(UCHAR *pByte);
extern void SerialByteSend(UCHAR byte);
extern void SendDebug(char value);
extern UINT16 Calc_CRC(UCHAR *ptr, UINT16 length);
extern UINT16 UINT16_Get(UCHAR *ptr);
extern void UINT16_Put(UCHAR *ptr, UINT16 value);


UCHAR ferr = 0;
UCHAR oerr = 0;

void SerialIF_Init(void){

    // disable the serial port while it is being configured
    RC1STAbits.SPEN = 0;

    // setup port for 8 bit, 9600 baud, no parity, 1 stop bit
    SP1BRGH = 0;
    SP1BRGL = 51;
    BAUD1CONbits.BRG16 = 0;
    TX1STAbits.BRGH = 0;
    TX1STAbits.TX9 = 0;

    // enable the async serial port
    TX1STAbits.SYNC = 0;

    // make sure serial port is in the enabled state
    EnableSerial(true);
}

/* Enable/Disable serial port feature
 *
 * Because all of the pins on the PIC are used when accessing the FDC board
 * it is necessary to "share" two pins when the serial port is used.
 *
 * In this case RC0 and RC1 become the Tx and Rx lines when using the serial port.
 *
 * Care must be taken to ONLY enable the serial feature when in an idle state
 * with the FDC board.  In other words you CAN'T use the serial port WHILE
 * accessing the FDC board.
 *
 * This restriction should not be an issue because the design is such that an
 * external host will:
 *
 *    1) serially send a command to the PIC
 *    2) wait while the PIC is performing the command
 *    3) when the PIC is done accessing the FDC board,
 *         it will send the reply to the host
 */
void EnableSerial(bool enabled){

    if (enabled){

        // force RC0 and RC1 ports to inputs so digital functions will work
        TRISCbits.TRISC0 = 1;
        TRISCbits.TRISC1 = 1;

        // re-map the RX1 input function to RC1
        RX1PPSbits.PORT = PPS_PORTC;                    // IMPORTANT! MUST be "PPS_PORTC" and NOT "PORTC" - THIS HAS WASTED MORE OF MY TIME THAN ANYTHING!
        RX1PPSbits.PIN = 1;

        // re-map the TX1 output function to RC0
        RC0PPS = PPS_TX1CK1;

        // enable the serial port now
        RC1STAbits.SPEN = 1;

#if 0
        // enable Rx interrupts
        PIE1bits.RC1IE = 1;
```

```
        INTCONbits.PEIE = 1;

#else
        // make sure Rx interrupts are disabled
        PIE1bits.RC1IE = 0;
#endif

        // enable the serial port lines on to PORTC d0/d1
        LATBbits.LATB4 = 0;

        // enable the transmitter
        TX1STAbits.TXEN = 1;

        // enable the receiver
        RC1STAbits.CREN = 1;

    }else{

        // disable the receiver
        RC1STAbits.CREN = 0;

        // disable the transmitter
        TX1STAbits.TXEN = 0;

        // disable Rx interrupts
        PIE1bits.RC1IE = 0;

        // disable the serial port
        RC1STAbits.SPEN = 0;

        // disable the serial port lines on to PORTC d0/d1
        // and restore PORTC for use talking to the FDC board
        LATBbits.LATB4 = 1;

        // re-map RC0 (Tx Serial Output) output functionality to be a normal port pin again
        RC0PPS = PPS_LATxy;

        // the UART Rx input will 'stay' connected to RC1 but since the receiver
        // is disabled, changes on RC1 will not affect it.

        // there is no need to disconnect the UART Rx functionality as all port
        // reads (on PORTC in this case) read the port pin values - NOT the
        // attached digital functionality.

    }
}

unsigned int debugCounter = 0;

/* SerialByteAvailable
 *
 * Get the next Rx byte if available
 */
bool SerialByteAvailable(UCHAR *pByte){

    bool available = false;

    if (PIR1bits.RC1IF){
        *pByte = RC1REG;
        available = true;
    }

    if (RC1STAbits.FERR){
        ferr++;
    }

    if (RC1STAbits.OERR){
        oerr++;

        // clear overrun error
        RC1STAbits.CREN = 0;
        RC1STAbits.CREN = 1;

    }

    return available;
}

/* SerialByteSend
 *
 * Send a serial byte and wait for it to leave the Tx buffer
 */
void SerialByteSend(UCHAR byte){
```

```
    TX1REG = byte;

    // wait until the character has been transmitted
    while (!TX1STAbits.TRMT);
}


#if 0
/* Send a debug character out the serial port
 *
 * This is extremely valuable when trying to debug power up timing issues
 * when there is not enough time to connect the debugger to see what is
 * going on.  This wouldn't be necessary if there were any spare debug pins
 * on the PIC, but there aren't.
 */
void SendDebug(char value){

    EnableSerial(true);

    TX1REG = value;

    // wait until the character has been transmitted
    while (!TX1STAbits.TRMT);

    EnableSerial(false);
}
#endif


// use the largest host structure as a receive buffer
UCHAR HostBuffer[sizeof(FloppyWriteStruct)];
UCHAR RxIdx = 0;
UCHAR RxCmdLen;

#define USE_TIMER   1

/* Process_Host_Commands
 *
 * Process commands from the host over the serial port.  This should be called
 * frequently (faster than serial bytes arrive) in the main loop.
 */
void Process_Host_Commands(void){


    static PRECISIONTIMER_TYPE timer;

    if (SerialByteAvailable(&HostBuffer[RxIdx])){

        timer = PrecisionTimerRead();

        switch (RxIdx++){
            case 0:
                switch (HostBuffer[0]){
                    case FLOPPY_CMD_FORMAT:     /* format drive */
                        RxCmdLen = sizeof(FloppyFormatStruct);
                        break;
                    case FLOPPY_CMD_READ:       /* read sector */
                        RxCmdLen = sizeof(FloppyReadStruct);
                        break;
                    case FLOPPY_CMD_WRITE:      /* write sector */
                        RxCmdLen = sizeof(FloppyWriteStruct);
                        break;
                    case FLOPPY_CMD_STATUS:     /* get status */
                        RxCmdLen = sizeof(FloppyStatusStruct);
                        break;
                    case FLOPPY_CMD_IDLE:       /* idle the physical drives */
                        RxCmdLen = sizeof(FloppyIdleStruct);
                        break;

                    default:
                        // invalid command, reset and start looking for another
                        RxIdx = 0;
                        break;
                }
                break;
            default:
                // has a full packet arrived
                if (RxIdx == RxCmdLen){

                    UINT16 crcRx = UINT16_Get(&HostBuffer[RxCmdLen - sizeof(((pFloppyFormatStruct) 0)->crc)]);
```

```
                if (Calc_CRC(HostBuffer, RxCmdLen - sizeof(((pFloppyFormatStruct) 0)->crc)) == crcRx){

                    UCHAR replyLen = 0;


                    switch (HostBuffer[0]){

                        case FLOPPY_CMD_FORMAT:     /* format drive */
                            {
                                pFloppyFormatStruct pFormat = (pFloppyFormatStruct) HostBuffer;
                                pFloppyFormatReplyStruct pFormatReply = (pFloppyFormatReplyStruct) HostBuffer;

                                pFormatReply->status = FP_STATUS_ERROR;

                                if (Drive[pFormat->drive].present){
                                    EnableSerial(false);

                                    if (Drive_Select(pFormat->drive)){
                                        pFormatReply->status = Format_Drive() ? FP_STATUS_OK : FP_STATUS_ERROR;
                                    }

                                    EnableSerial(true);
                                }
                                UINT16_Put((PUCHAR) &pFormatReply->crc, Calc_CRC(HostBuffer, sizeof(*pFormatReply) - sizeof(pFormatReply->crc)));

                                replyLen = sizeof(*pFormatReply);
                            }
                            break;

                        case FLOPPY_CMD_READ:       /* read sector */
                            {
                                pFloppyReadStruct pRead = (pFloppyReadStruct) HostBuffer;
                                pFloppyReadReplyStruct pReadReply = (pFloppyReadReplyStruct) HostBuffer;

                                pReadReply->status = STATUS_DRIVE_SELECT_FAIL;

                                if (Drive[pRead->drive].present){
                                    EnableSerial(false);

                                    if (Drive_Select((DRIVE_SELECTED_TYPE) pRead->drive)){
                                        pReadReply->status = Read_Sector(pRead->track, pRead->sector, pReadReply->buffer);
                                    }

                                    if (pReadReply->status != STATUS_SUCCESS){
                                        // fill return buffer with error code

                                        for (int x=0; x<NUM_ENTRIES(HostBuffer); x++){
                                            HostBuffer[x] = pReadReply->status;
                                        }
                                    }

                                    EnableSerial(true);
                                }
                                UINT16_Put((PUCHAR) &pReadReply->crc, Calc_CRC(HostBuffer, sizeof(*pReadReply) - sizeof(pReadReply->crc)));

                                replyLen = sizeof(*pReadReply);
                            }
                            break;

                        case FLOPPY_CMD_WRITE:      /* write sector */
                            {
                                pFloppyWriteStruct pWrite = (pFloppyWriteStruct) HostBuffer;
                                pFloppyWriteReplyStruct pWriteReply = (pFloppyWriteReplyStruct) HostBuffer;

                                pWriteReply->status = STATUS_DRIVE_SELECT_FAIL;

                                if (Drive[pWrite->drive].present){
                                    EnableSerial(false);

                                    if (Drive_Select((DRIVE_SELECTED_TYPE) pWrite->drive)){
                                        pWriteReply->status = Write_Sector(pWrite->track, pWrite->sector, pWrite->buffer);
                                    }
                                    EnableSerial(true);
                                }
                                UINT16_Put((PUCHAR) &pWriteReply->crc, Calc_CRC(HostBuffer, sizeof(*pWriteReply) - sizeof(pWriteReply->crc)));

                                replyLen = sizeof(*pWriteReply);
                            }
                            break;
                        case FLOPPY_CMD_STATUS:     /* get status */
                            {
                                pFloppyStatusReplyStruct pStatusReply = (pFloppyStatusReplyStruct) HostBuffer;
// TODO USE REAL DATA
```

```
// TODO USE REAL DATA
                                    //EnableSerial(false);

                                    pStatusReply->controllerStatus = uPD372D_Functional() ? 0x00 : 0x01;
                                    for (int x=0; x<NUM_ENTRIES(pStatusReply->driveStatus); x++){
                                        pStatusReply->driveStatus[x] = 0x00;
                                    }

                                    //EnableSerial(true);

                                    replyLen = sizeof(*pStatusReply);
                                }
                                break;
                            case FLOPPY_CMD_IDLE:      /* idle drives */
                                {
                                    pFloppyIdleReplyStruct pIdleReply = (pFloppyIdleReplyStruct) HostBuffer;

                                    EnableSerial(false);

                                    pIdleReply->status = Drive_Select(NoDriveSelected) ? FP_STATUS_OK : FP_STATUS_ERROR;

                                    EnableSerial(true);

                                    UINT16_Put((PUCHAR) &pIdleReply->crc, Calc_CRC(HostBuffer, sizeof(*pIdleReply) - sizeof(pIdleReply->crc)));

                                    replyLen = sizeof(*pIdleReply);
                                }
                                break;

                            default:
                                break;
                        }

                        // send the reply out to the host
                        for (int x=0; x<replyLen; x++){
                            SerialByteSend(HostBuffer[x]);
                        }

                        // start looking for another new packet
                        RxIdx = 0;
                    }
                }
                break;
        }
    }else{

        // @9600 each character should take about 1mS to arrive
        if (PrecisionTimeruSDelta(timer) > 10000){
            // timed out waiting for the next character,
            // reset state machine to start looking for a new packet
            RxIdx = 0;
        }
    }
}


/* Calc_CRC
 *
 * Calculate CRC on buffer start at *ptr for length #of bytes
 */
UINT16 Calc_CRC(UCHAR *ptr, UINT16 length) {

    UINT16 crc = 0;
UINT16 nextchar;
UINT16 temp;

for (; length > 0; length--) {
nextchar = *ptr++;
temp = (crc ^ nextchar) & 0x0f;
crc = (crc >> 4) ^ (temp * 0x1081);
temp = (crc ^ (nextchar >> 4)) & 0x0f;
crc = (crc >> 4) ^ (temp * 0x1081);
}
return(crc);
}

/* UINT16_Get
 *
 * Get a 16 bit value from a packet with protocol byte order
 */
UINT16 UINT16_Get(UCHAR *ptr){
```

```
    return *ptr + (*(ptr + 1) * 0x100);
}

/* UINT16_Put
 *
 * Put a 16 bit value into a packet with protocol byte order
 */
void UINT16_Put(UCHAR *ptr, UINT16 value){

    *ptr = (UCHAR) (value % 0x100);
    *(ptr + 1) = value / 0x100;
}
```

```
#include <xc.h>
#include <pic16f15244.h>

#include "Common.h"
#include "Timer.h"

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



// Timer Initialization
void Timer_Init(void){

#if SUPPORT_TIMER0
    // 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
#endif

#if SUPPORT_TIMER1
    // Initialize Timer1 to be a free running 16bit hardware 1.024mS counter
    // 32KHz / 8 (prescaler) = 4KHz (0.250mS)
    // REMEMBER to update PrecisionTimeruSPeriod() if you change this
    T1GCONbits.GE = 0;          // disable gating feature so it is free running
//    T1CLK = 3;                  // HFINTOSC (see config bits, usually RSTOSC = HFINTOSC_32MHZ)
    T1CLK = 7;                  // MFINTOSC (32KHz)
    T1CONbits.CKPS = 3;         // 1:8 prescaler
    T1CONbits.RD16 = 1;         // enable 16bit r/w operations
    T1CONbits.TMR1ON = 1;       // enable timer1
#endif



#if SUPPORT_TIMER2

    #define T2TIMER_UPPER_LIMIT     250

    // Initialize Timer2 to be a free running 8 bit timer
    // 32KHz / 128 = 250Hz (4mS period)
    T2CLKCONbits.CS = 0x05;     // select MFINTOSC (32KHz) clock

    T2HLTbits.MODE = 0x00;      // setup mode to be "free running, period pulse, software gate"

    T2PR = T2TIMER_UPPER_LIMIT; // setup to have counter roll over at T2TIMER_UPPER_LIMIT (i.e. time events up to T2TIMER_UPPER_LIMIT x 4mS)

    T2CONbits.CKPS = 0x07;      // 1:128 prescaler
    T2CONbits.ON = 1;
#endif
}

#if SUPPORT_TIMER0
// 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) {

       // clear this interrupt condition
```

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


#if SUPPORT_TIMER1
// read precision timer value
inline PRECISIONTIMER_TYPE PrecisionTimerRead(void){

    // 16 bit mode requires that TMR1L be read first
    // this ensures that the corresponding TMR1H value
    // is latched into a temp register at the time of
    // the TMR1L read.  When TMR1H is later read, it
    // will be this latched value, and NOT the current
    // high byte of the free running timer.
    PRECISIONTIMER_TYPE value = TMR1L;

    return value + (TMR1H * 0x100);
}

// get the precision timer uS period
inline const PRECISIONTIMER_TYPE PrecisionTimeruSPeriod(void){
    return 250;
}

#if 0
// get the max time period you can measure with the Precision Timer
// due to hardware and software constraints
inline const PRECISIONTIMER_TYPE PrecisionTimerMaxuSLimit(void){
    return ((PRECISIONTIMER_TYPE) -1) * PrecisionTimeruSPeriod();

    // if PrecisionTimeruSDelta were changed to return a ULONG
    // instead of PRECISIONTIMER_TYPE, more precision could be
    // attained.  However this would double the memory requirements
    // for the variables so it has been deemed better to limit
    // the resolution to 65535uS which should be reasonable for
    // most precision measurements.
    //return ((PRECISIONTIMER_TYPE) -1) * PrecisionTimeruSPeriod;
}
#endif

// calculate the time delta from the previous PrecisionTimerRead() value
// TODO deal with possible overflow when converting to uS
inline PRECISIONTIMER_TYPE PrecisionTimeruSDelta(PRECISIONTIMER_TYPE prevRead){

    PRECISIONTIMER_TYPE nowRead = PrecisionTimerRead();

    if (nowRead >= prevRead){
        return (nowRead - prevRead) * PrecisionTimeruSPeriod();
    } else {
        return ((((PRECISIONTIMER_TYPE) -1) - prevRead) + nowRead) * PrecisionTimeruSPeriod();
    }
}
#endif


#if SUPPORT_TIMER2
inline MSTIMER_TYPE mSTimerRead(void){

    return T2TMR;
}

// get the precision timer mS period
inline const MSTIMER_TYPE mSTimerPeriod(void){
    return 4;
}

// calculate the time delta from the previous PrecisionTimerRead() value
// TODO deal with possible overflow when converting to uS
inline UINT16 mSTimerDelta(MSTIMER_TYPE prevRead){

    MSTIMER_TYPE nowRead = T2TMR;
```

```
    if (nowRead >= prevRead){
        return (nowRead - prevRead) * mSTimerPeriod();
    } else {
        return ((((UINT16) T2TIMER_UPPER_LIMIT) - prevRead) + nowRead) * mSTimerPeriod();
    }
}
#endif
```