```
/*
 * File:   KbdDriver.h
 * Author: Alec
 *
 * Created on July 26, 2021, 5:12 PM
 */

#ifndef KBDDRIVER_H
#defineKBDDRIVER_H

#ifdef__cplusplus
extern "C" {
#endif

#include <stdbool.h>

// Keyboard Protocol works as follows on key press:
//
//     ExtendedScanCode, make code                  (ex: Up Arrow pressed -> ExtendedScanCode, 0x75)
//     make code                                    (ex: the letter 'A' pressed -> 0x1c)
//
// Keyboard Protocol works as follows on key release:
//
//     ExtendedScanCode, BreakScanCode, make code   (ex: Up Arrow released -> ExtendedScanCode, BreakScanCode, 0x75)
//     BreakScanCode, make code                     (ex: the letter 'A' released -> BreakScanCode, 0x1c)
//
#define ExtendedScanCode    0xe0
#define BreakScanCode       0xf0


// states used to keep track of the byte stream from the keyboard
typedef enum {
    SETLEDCMD = 0,          // set LED state command (may not be supported in older keyboards)
    SETLEDDATA,             // configuring keyboard LED state, 2nd byte sent after SETLEDCMD state
    SINGLEMAKE,             // waiting for 1st byte of packet to determine packet type
    EXTENDEDMAKE,           // ExtendedScanCode rxd, waiting for 2nd byte to determine if press or release
    BREAKCODE,              // BreakScanCode rxd (without ExtendedScanCode), waiting for 2nd byte to determine key released
    EXTENDEDBREAK           // ExtendedScanCode followed by BreakScanCode rxd, waiting for 3rd byte to determine extended key released
} KEYBOARD_BYTE_STATE;

extern bool KeyboardDetected;

extern bool HardwareFailure;

extern KEYBOARD_BYTE_STATE KbdByteState;

// Keyboard Driver initialization
void KbdDriver_Init(void (*Upper_ProcessScanCode)(unsigned char scanCode));
// IOC Interrupt handler
void Keyboard_IOC_Handler(void);
// Timer Interrupt handler
void Keyboard_Timer_Handler(void);
// change the keyboard LED state
void Keyboard_Set_LEDs(bool numLock, bool capLock, bool scrollLock);



#ifdef__cplusplus
}
#endif

#endif/* KBDDRIVER_H */
```

```
/*
 * File:   Keyboard.h
 * Author: Alec
 *
 * Created on July 21, 2021, 9:04 AM
 */

#ifndef KEYBOARD_H
#defineKEYBOARD_H

#ifdef __cplusplus
extern "C" {
#endif

#include <stdbool.h>

#include "KbdDriver.h"

extern bool CapsLockActive;             // true if the caps lock is active
extern bool NumLockActive;              // true if the num lock is active
extern bool ScrollLockActive;           // true if the scroll lock is active

// Initialization logic
void Keyboard_Init(void);

// key sequences that have no ASCII equivalent that are ARE exposed to the caller
typedef enum {
    INSERT_KEY = 0x90,
    UARROW_KEY,
    DARROW_KEY,
    LARROW_KEY,
    RARROW_KEY,
    F1K,
    F2K,
    F3K,
    F4K,
    F5K,
    F6K,
    F7K,
    F8K,
    F9K,
    F10K,
    F11K,
    F12K,
    LAST_ASCII_EXTENSION
} KEYBOARD_ASCII_EXTENSIONS;

// Keypressed will be either one of the normal ASCII character set or one of KEYBOARD_ASCII_EXTENSIONS
// ControlActive will be true if the 'control' key was held down when this key was pressed
// AltActive will be true if the 'alt' key was held down when this key was pressed
void Keyboard_Register_Callback(void (*callback)(unsigned char Keypressed , bool ControlActive, bool AltActive));


#ifdef __cplusplus
}
#endif

#endif/* KEYBOARD_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);

// 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



#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

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

#ifndef TIMER_H
#defineTIMER_H

#ifdef__cplusplus
extern "C" {
#endif

#define Timer0uSPeriod      7936
#define Timer2uSPeriod      256

// Timer Initialization
void Timer_Init(void);

// Timer interrupt handler
void Timer_Handler(void);

// Start Timer and wait for interrupt
// Valid range for uSeconds is: 256uS (Timer2uSPeriod) - 65,280uS (Timer2uSPeriod * 255)
// (values outside this range will be clamped to the range limits)
void Start_Timer2(unsigned int uSeconds);

// disable the timeout if it is no longer needed
void Stop_Timer2(void);

// free running tick counter;
extern unsigned char TickCounter;

#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]))



#ifdef__cplusplus
}
#endif

#endif/* UTILITY_H */
```

```
/*
 * File:   Keyboard.c
 * Author: Alec
 *
 * Created on July 26, 2021, 5:13 PM
 */

#include <xc.h>

#include "KbdDriver.h"
#include "Timer.h"

bool KeyboardDetected = false;
bool HardwareFailure = false;

// This logic assumes that the following keyboard signals are connected to the specified PIC pins
// RB5 - Clock
// RB6 - Data

// Clock and Data can be in the following states
typedef enum{
    LINE_LOW = 0,
    LINE_HIGH,
    LINE_TRISTATE
} LINE_STATE;

// time to drive clock low when initiating a transmission
// (spec says anything over 100uS is valid - since this is too
//  short for our timer logic it will be bumped up to Timer2uSPeriod
//  (256uS))
#define TX_INITIATE_TIMEOUT_US      150
// max time to wait for keyboard to generate first clock when transmitting
#define TX_KYBD_CLK_TIMEOUT_US      15000
// max time to send/receive a full byte
#define TX_KYBD_BYTE_TIMEOUT_US     2000


// states used to send/receive the bit stream to/from the keyboard
static enum {
    // Rx bit states used when waiting for data from the keyboard
    RX_STARTBIT = 0,

    RX_D0BIT,
    RX_D1BIT,
    RX_D2BIT,
    RX_D3BIT,
    RX_D4BIT,
    RX_D5BIT,
    RX_D6BIT,
    RX_D7BIT,
    RX_PARITYBIT,
    RX_STOPBIT,

    // Tx bit states used when sending data to the keyboard
    TX_INITIATE,                // Drive clk low for TX_INITIATE_TIMEOUT_US, then drive data low and release clk line
    TX_WAIT_CLK_LOW,            // Wait for keyboard to drive clk low
    TX_STARTBIT,
    TX_D0BIT,                   //   D0-Parity then sent as keyboard sends each clock pulse out
    TX_D1BIT,
    TX_D2BIT,
    TX_D3BIT,
    TX_D4BIT,
    TX_D5BIT,
```

```
    TX_D6BIT,
    TX_D7BIT,
    TX_PARITYBIT,
    TX_WAIT_DATA_LOW,           // Wait for keyboard to drive data low
    TX_COMPLETE,
    TX_ERROR

} KbdBitState;


// Keyboard Power Up sequence (may not be present on very old keyboards)
//
//      Keyboard                        Host
//  ------------------          ------------------------
//  Kybd_BAT_Passed/Failed ->
//                          <-  SetReset_Status
//  Kybd_Ack                ->
//                          <-  LED status bits (0x00 = all off)
//  Kybd_Ack                ->
//

//  Typical PC startup includes more configuration, but for our purposes
//  these are not needed, the keyboard will start sending keys without them.
//
//  Note that modern keyboards require the host to acknowledge the BAT results
//  in this manner.  Failure to do so will cause the keyboard to keep re-sending
//  the BAT results over and over again.

#define Kybd_BAT_Passed     0xaa
#define Kybd_BAT_Failed     0xfc
#define SetLED_Cmd          0xed
#define Kybd_Ack            0xfa
#define Kybd_Resend         0xfe        /* keyboard sends this if you send it an unrecognized command */
// 'or' the following command byte with the bits below to turn on specific LEDs
#define Set_LED_Command     0x00
    #define LED_Scroll_Bit      0x01
    #define LED_Num_Bit         0x02
    #define LED_Caps_Bit        0x04

// states used to keep track of the byte stream from the keyboard
KEYBOARD_BYTE_STATE KbdByteState;

static unsigned char KeyboardData;      // assembly area for incoming/outgoing bits
static unsigned char KeyboardData2;     // optional 2nd byte of data to send to keyboard
static bool ParityBit;

static void Drive_Data_Line(LINE_STATE state);
static void Drive_Clock_Line(LINE_STATE state);
static bool Calc_Parity(void);
static void ProcessScanCode(unsigned char scanCode);
void Keyboard_Initiate_SetLED_State(void);


static void (*HighLevel_ProcessScanCode)(unsigned char scanCode);

// initialization logic
void KbdDriver_Init(void (*Upper_ProcessScanCode)(unsigned char scanCode)){

    // store the high level scan code handler
    HighLevel_ProcessScanCode = Upper_ProcessScanCode;

    // Set Clock line as an input, disable slew rate limitation to improve interrupt response
```

```
    ODCONBbits.ODCB5 = 1;       // set port bit to operate as Open-Drain
    Drive_Clock_Line(LINE_TRISTATE);
    SLRCONBbits.SLRB5 = 0;      // slew adds about 140uS to the IOC falling edge response

    // Set Data line as an input, disable slew rate limitation to improve interrupt response
    ODCONBbits.ODCB6 = 1;
    Drive_Data_Line(LINE_TRISTATE);
    SLRCONBbits.SLRB6 = 0;      // slew adds about 140uS to the IOC falling edge response

    // Enable interrupt on falling edge of clock
    IOCBPbits.IOCBP5 = 0;
    IOCBNbits.IOCBN5 = 1;

    // initialize keyboard state machine
    KbdBitState = RX_STARTBIT;
    KbdByteState = SINGLEMAKE;
    KeyboardData = 0x00;

    // enable interrupts on change of pin states
    PIE0bits.IOCIE = 1;
}

// Drive the Clock line to a specific state
static void Drive_Clock_Line(LINE_STATE state){

    switch (state){
        case LINE_LOW:
            // enable weak pull-up
            WPUBbits.WPUB5 = 1;
            // Set Clock state low
            PORTBbits.RB5 = 0;
            // Set Clock line as an output
            TRISBbits.TRISB5 = 0;
            break;

        case LINE_HIGH:
            // enable weak pull-up
            WPUBbits.WPUB5 = 1;
            // Set Clock state high
            PORTBbits.RB5 = 1;
            // Set Clock line as an output
            TRISBbits.TRISB5 = 0;
            break;

        default:
        case LINE_TRISTATE:
            // enable weak pull-up
            WPUBbits.WPUB5 = 1;
            // Set Clock state high
            PORTBbits.RB5 = 1;
            // Set Clock line as an input
            TRISBbits.TRISB5 = 1;
            break;
    }
}

// Drive the Data line to a specific state
static void Drive_Data_Line(LINE_STATE state){

    switch (state){
        case LINE_LOW:
            // enable weak pull-up
```

```
            WPUBbits.WPUB6 = 1;
            // Set Data state low
            PORTBbits.RB6 = 0;
            // Set Data line as an output
            TRISBbits.TRISB6 = 0;
            break;

        case LINE_HIGH:
            // enable weak pull-up
            WPUBbits.WPUB6 = 1;
            // Set Data state high
            PORTBbits.RB6 = 1;
            // Set Data line as an output
            TRISBbits.TRISB6 = 0;
            break;

        default:
        case LINE_TRISTATE:
            // enable weak pull-up
            WPUBbits.WPUB6 = 1;
            // Set Data state high
            PORTBbits.RB6 = 1;
            // Set Data line as an input
            TRISBbits.TRISB6 = 1;
            break;
    }
}

// parity is set if there are an even number of '1's in the data byte (even parity)
static bool Calc_Parity(void){

    unsigned char data = KeyboardData;
    int ones = 0;
    for (int x=0; x<8; x++){
        if (data & 0x01) ones++;
        data = data >> 1;
    }

    return ((ones & 0x0001) ? false : true);
}

// change the keyboard LED state
void Keyboard_Set_LEDs(bool numLock, bool capLock, bool scrollLock){

    // setup to tell the keyboard which LEDs should be on
    KeyboardData2 = Set_LED_Command
            | (numLock ? LED_Num_Bit : 0)
            | (capLock ? LED_Caps_Bit : 0)
            | (scrollLock ? LED_Scroll_Bit : 0);

    // setup state machine to send out first byte of set LED sequence
    Keyboard_Initiate_SetLED_State();
}

// setup state machine to send out first byte of set LED sequence
void Keyboard_Initiate_SetLED_State(void){

    // set the byte state to setting LED states
    KbdByteState = SETLEDCMD;

    // setup to tell the keyboard which LEDs should be on
```

```
    KeyboardData = SetLED_Cmd;

    // update the parity bit
    ParityBit = Calc_Parity();

    // indicate we are sending data now
    KbdBitState = TX_INITIATE;
    // and start the timer for next state transition in timer interrupt
    Start_Timer2(TX_INITIATE_TIMEOUT_US);

    // drive clock line low
    //
    // THIS MUST BE DONE AFTER KbdBitState HAS BEEN CHANGED BECAUSE
    // DRIVING THE CLOCK LINE LOW WILL ALSO TRIGGER A FALLING EDGE
    // INTERRUPT AND IF KbdBitState IS STILL IN THE RX_STOPBIT STATE
    // THEN IT WILL CAUSE THE STATE MACHINE TO START LOOKING FOR
    // ANOTHER BYTE FROM THE KEYBOARD - INSTEAD OF TRANSMITTING THE
    // BYTE TO THE KEYBOARD AS SHOULD BE DONE.
    Drive_Clock_Line(LINE_LOW);
}

// Keyboard interrupt handler
void Keyboard_IOC_Handler(void){

    static bool ValidStartBit = 0;
    static bool KybdAckRxd = 0;

    // if IOC interrupts are enabled AND active
    if (IOCIE && IOCIF){

        // if clock just went low
        if (IOCBFbits.IOCBF5){

            switch(KbdBitState){

                //------------------------------------------------------
                //  Receive States
                //------------------------------------------------------

                case RX_STARTBIT:                   // should be low
                    if (PORTBbits.RB6){

                        // indicate this byte is invalid
                        ValidStartBit = 0;
                    }else{
                        ValidStartBit = 1;
                        KbdBitState = RX_D0BIT;
                        KeyboardData = 0x00;
                    }
                    // start timer to make sure we get the full byte in the
                    // required time frame
                    Start_Timer2(TX_KYBD_BYTE_TIMEOUT_US);
                    break;
                case RX_D0BIT:
                    if (PORTBbits.RB6) KeyboardData |= 0x01;
                    KbdBitState = RX_D1BIT;
                    break;
                case RX_D1BIT:
                    if (PORTBbits.RB6) KeyboardData |= 0x02;
                    KbdBitState = RX_D2BIT;
                    break;
                case RX_D2BIT:
                    if (PORTBbits.RB6) KeyboardData |= 0x04;
```

```
                KbdBitState = RX_D3BIT;
                break;
            case RX_D3BIT:
                if (PORTBbits.RB6) KeyboardData |= 0x08;
                KbdBitState = RX_D4BIT;
                break;
            case RX_D4BIT:
                if (PORTBbits.RB6) KeyboardData |= 0x10;
                KbdBitState = RX_D5BIT;
                break;
            case RX_D5BIT:
                if (PORTBbits.RB6) KeyboardData |= 0x20;
                KbdBitState = RX_D6BIT;
                break;
            case RX_D6BIT:
                if (PORTBbits.RB6) KeyboardData |= 0x40;
                KbdBitState = RX_D7BIT;
                break;
            case RX_D7BIT:
                if (PORTBbits.RB6) KeyboardData |= 0x80;
                KbdBitState = RX_PARITYBIT;
                break;
            case RX_PARITYBIT:
                KbdBitState = RX_STOPBIT;
                break;
            case RX_STOPBIT:               // should be high
                // full byte received from keyboard, stop timeout
                Stop_Timer2();
                // reset state machine to process next byte from keyboard
                KbdBitState = RX_STARTBIT;

                // if the start and stop bits are valid
                if (ValidStartBit && PORTBbits.RB6){
                    // a full key scan has now been received
                    // MUST be after timer & state are reset above
                    ProcessScanCode(KeyboardData);
                }
                break;

            //------------------------------------------------------
            //  Transmit States
            //------------------------------------------------------
            case TX_STARTBIT:
                Drive_Data_Line(KeyboardData & 0x01 ? LINE_HIGH : LINE_LOW);

                // re-start the timeout to handle the case of byte took too long to send
                Start_Timer2(TX_KYBD_BYTE_TIMEOUT_US);

                KbdBitState = TX_D0BIT;
                break;
            case TX_D0BIT:
                Drive_Data_Line(KeyboardData & 0x02 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D1BIT;
                break;
            case TX_D1BIT:
                Drive_Data_Line(KeyboardData & 0x04 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D2BIT;
                break;
            case TX_D2BIT:
                Drive_Data_Line(KeyboardData & 0x08 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D3BIT;
```

```
                break;
            case TX_D3BIT:
                Drive_Data_Line(KeyboardData & 0x10 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D4BIT;
                break;
            case TX_D4BIT:
                Drive_Data_Line(KeyboardData & 0x20 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D5BIT;
                break;
            case TX_D5BIT:
                Drive_Data_Line(KeyboardData & 0x40 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D6BIT;
                break;
            case TX_D6BIT:
                Drive_Data_Line(KeyboardData & 0x80 ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_D7BIT;
                break;
            case TX_D7BIT:
                Drive_Data_Line(ParityBit ? LINE_HIGH : LINE_LOW);
                KbdBitState = TX_PARITYBIT;
                break;
            case TX_PARITYBIT:
                // Release the data line
                Drive_Data_Line(LINE_TRISTATE);

                // Wait for the data line to be driven low by the keyboard
                // before we do anything else.
                KbdBitState = TX_WAIT_DATA_LOW;
                break;

            case TX_WAIT_DATA_LOW:
                // Check to see if the keyboard drove the data line low
                // signifying that it is acknowledging our command
                if (!PORTBbits.RB6){
                    KybdAckRxd = true;
                }else{
                    KybdAckRxd = false;
                }

                // indicate transmission is complete
                KbdBitState = TX_COMPLETE;


                // abort the byte timeout as a full byte was transmitted
                Stop_Timer2();

                // reset the keyboard bit state to start looking for a new packet of data from the keyboard
                KbdBitState = RX_STARTBIT;
                break;


            case TX_INITIATE:
            case TX_WAIT_CLK_LOW:
            case TX_COMPLETE:
                // interrupts during these states should be ignored
                break;

            default:
                KbdBitState = RX_STARTBIT;
                break;
        }

        IOCBFbits.IOCBF5 = 0;
```

```
        }

        // RESET ALL OF THE UNUSED CHANGE FLAGS OTHERWISE THIS INTERUPT EATS ALL OF THE CPU TIME
        IOCAF &= 0x00;
        IOCBF &= 0x20;
        IOCCF &= 0x00;
    }
}

// Keyboard timer interrupt handler
void Keyboard_Timer_Handler(void){

    switch (KbdBitState){

        // ************************************************************
        // time to release the clock line so that the keyboard can start
        // generating the clock pulses
        // ************************************************************
        case TX_INITIATE:


            // update bit state to looking for keyboard clock for start bit
            KbdBitState = TX_STARTBIT;

            // Drive Data low for start bit
            Drive_Data_Line(LINE_LOW);

            // Release the clock line
            Drive_Clock_Line(LINE_TRISTATE);

            // re-start the timeout to handle the case of no keyboard clocks ever received
            Start_Timer2(TX_KYBD_CLK_TIMEOUT_US);

            // let the clock interrupt handle the remaining bits
            break;

        // ************************************************************
        // transmission was requested but keyboard didn't generate
        // the first clock pulse in the required window
        // ************************************************************
        case TX_STARTBIT:

            // release the data line since we are in an error condition
            Drive_Data_Line(LINE_TRISTATE);

            // revert to looking for key presses
            KbdByteState = SINGLEMAKE;
            KbdBitState = RX_STARTBIT;
            break;

        // ************************************************************
        // transmission started but it took too long for the keyboard to
        // generate all of the required clock pulses
        // ************************************************************
        case TX_D0BIT:
        case TX_D1BIT:
        case TX_D2BIT:
        case TX_D3BIT:
        case TX_D4BIT:
        case TX_D5BIT:
        case TX_D6BIT:
        case TX_D7BIT:
```

```
        case TX_PARITYBIT:

            // release the data line since we are in an error condition
            Drive_Data_Line(LINE_TRISTATE);

            // revert to looking for key presses
            KbdByteState = SINGLEMAKE;
            KbdBitState = RX_STARTBIT;
            break;

        // ************************************************************
        // timed out receiving bits from the keyboard before a
        // full byte could be assembled
        // ************************************************************
        case RX_STARTBIT:
            // nothing to do here, just keep looking
            break;
        case RX_D0BIT:
        case RX_D1BIT:
        case RX_D2BIT:
        case RX_D3BIT:
        case RX_D4BIT:
        case RX_D5BIT:
        case RX_D6BIT:
        case RX_D7BIT:
        case RX_PARITYBIT:
        case RX_STOPBIT:
            // reset the keyboard bit state to start looking for a new packet of data from the keyboard
            KbdBitState = RX_STARTBIT;
            break;

        case TX_COMPLETE:
        default:
            break;
    }
}

// update our state machine with the scan code just received
static void ProcessScanCode(unsigned char scanCode){

    unsigned char key;


    switch (scanCode){

        case Kybd_BAT_Passed:
        case Kybd_BAT_Failed:

            KeyboardDetected = true;
            HardwareFailure = (scanCode == Kybd_BAT_Failed) ? true : false;

            // on power up always turn the LEDs off - assume no 'lock' states exist yet
            KeyboardData2 = 0x00;

            // setup state machine to send out first byte of set LED sequence
            Keyboard_Initiate_SetLED_State();

            break;

        case Kybd_Ack:
            switch (KbdByteState){

                case SETLEDCMD:
```

```
                    // set the byte state to setting LED states
                    KbdByteState = SETLEDDATA;

                    // tell the keyboard which LEDs should be on
                    KeyboardData = KeyboardData2;

                    // update the parity bit
                    ParityBit = Calc_Parity();

                    // indicate we are sending data now
                    KbdBitState = TX_INITIATE;
                    // and start the timer for next state transition in timer interrupt
                    Start_Timer2(TX_INITIATE_TIMEOUT_US);
                    break;

                default:
                case SETLEDDATA:

                    // ack rx'd from keyboard, stop time out
                    Stop_Timer2();

                    // set the byte state to waiting for key scan data
                    KbdByteState = SINGLEMAKE;
                    break;
            }
            break;

        default:
            // If there is a valid high level scan code processor
            if (HighLevel_ProcessScanCode != NULL){
                // invoke it so the other scan code cases can be handled
                HighLevel_ProcessScanCode(scanCode);
            }
            break;
    }
}
```

```
/*
 * File:   Keyboard.c
 * Author: Alec
 *
 * Created on July 21, 2021, 9:04 AM
 */

#include <xc.h>

#include "Keyboard.h"
#include "KbdDriver.h"

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

static bool AltActive;                  // true if the alt key is held down
static bool ControlActive;              // true if the control key is held down
static bool ShiftActive;                // true if a shift key is held down

bool CapsLockActive;             // true if the caps lock is active
bool NumLockActive;              // true if the num lock is active
bool ScrollLockActive;           // true if the scroll lock is active

// user callback to receive each key as it's pressed
static void (*Keyboard_Callback)(unsigned char Keypressed, bool ControlActive, bool AltActive);

// update our state machine with the scan code just received
static void ProcessScanCode(unsigned char scanCode);
// keyPress is mostly ASCII - with KEYBOARD_ASCII_EXTENSIONS
static void HandleKeyPress(unsigned char keyPress);

// initialization logic
void Keyboard_Init(void){

    KbdDriver_Init(ProcessScanCode);

    ShiftActive = ControlActive = AltActive = false;
    NumLockActive = CapsLockActive = ScrollLockActive = false;

    Keyboard_Callback = NULL;
}


// allow the caller to register a callback to handle key presses
void Keyboard_Register_Callback(void (*callback)(unsigned char Keypressed, bool ControlActive, bool AltActive)){

    Keyboard_Callback = callback;
}

// key sequences that have no ASCII equivalent that are NOT exposed to the caller
// (keep names short - 4 char - so they fit in the ScanCode arrays nicely)
typedef enum{
    CPLK = LAST_ASCII_EXTENSION,    /* caps lock */
    LSHT,                           /* left shift */
    RSHT,                           /* right shift */
    LCTL,                           /* left control */
    RCTL,                           /* right control */
    LALT,                           /* left alt */
    RALT,                           /* right alt */

    UIMP = 0xff        /* unimplemented scan code */

} KEYBOARD_LOCAL_ASCII_EXTENSIONS;

// non-shifted scan code conversion table
static const unsigned char ScanCodes[] = {
```

```
//   0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
   UIMP,  F9K, UIMP,  F5K,  F3K,  F1K,  F2K, F12K, UIMP, F10K,  F8K,  F6K,  F4K, 0x09,  '`', UIMP,     // 0x0x
   UIMP, LALT, LSHT, UIMP, LCTL,  'q',  '1', UIMP, UIMP, UIMP,  'z',  's',  'a',  'w',  '2', UIMP,     // 0x1x
   UIMP,  'c',  'x',  'd',  'e',  '4',  '3', UIMP, UIMP,  ' ',  'v',  'f',  't',  'r',  '5', UIMP,     // 0x2x
   UIMP,  'n',  'b',  'h',  'g',  'y',  '6', UIMP, UIMP, UIMP,  'm',  'j',  'u',  '7',  '8', UIMP,     // 0x3x
   UIMP,  ',',  'k',  'i',  'o',  '0',  '9', UIMP, UIMP,  '.', UIMP,  'l',  ';',  'p',  '-', UIMP,     // 0x4x
   UIMP, UIMP, UIMP, UIMP,  '[',  '=', UIMP, UIMP, CPLK, RSHT, 0x0d,  ']', UIMP, '\\', UIMP, UIMP,     // 0x5x
   UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, 0x08, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP,     // 0x6x
   UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, 0x27, F11K,  '+', UIMP,  '-',  '*', UIMP, UIMP, UIMP,     // 0x7x
   UIMP, UIMP, UIMP,  F7K, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP      // 0x8x
};

// shifted scan code conversion table
static const unsigned char ShiftedScanCodes[] = {
//   0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
   UIMP,  F9K, UIMP,  F5K,  F3K,  F1K,  F2K, F12K, UIMP, F10K,  F8K,  F6K,  F4K, 0x09,  '~', UIMP,     // 0x0x
   UIMP, LALT, LSHT, UIMP, LCTL,  'Q',  '!', UIMP, UIMP, UIMP,  'Z',  'S',  'A',  'W',  '@', UIMP,     // 0x1x
   UIMP,  'C',  'X',  'D',  'E',  '$',  '#', UIMP, UIMP,  ' ',  'V',  'F',  'T',  'R',  '%', UIMP,     // 0x2x
   UIMP,  'N',  'B',  'H',  'G',  'Y',  '^', UIMP, UIMP, UIMP,  'M',  'J',  'U',  '7',  '*', UIMP,     // 0x3x
   UIMP,  '<',  'K',  'I',  'O',  ')',  '(', UIMP, UIMP,  '>', UIMP,  'L',  ':',  'P',  '_', UIMP,     // 0x4x
   UIMP, UIMP, UIMP, UIMP,  '{',  '+', UIMP, UIMP, CPLK, RSHT, 0x0d,  '}', UIMP,  '|', UIMP, UIMP,     // 0x5x
   UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, 0x08, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP,     // 0x6x
   UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, 0x27, F11K,  '+', UIMP,  '-',  '*', UIMP, UIMP, UIMP,     // 0x7x
   UIMP, UIMP, UIMP,  F7K, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP, UIMP      // 0x8x
};


// keyPress is mostly ASCII - with KEYBOARD_ASCII_EXTENSIONS
static void HandleKeyPress(unsigned char keyPress){

    switch (keyPress){
        case CPLK:          /* caps lock */
        case LSHT:          /* left shift */
        case RSHT:          /* right shift */
        case LCTL:          /* left control */
        case RCTL:          /* right control */
        case LALT:          /* left alt */
        case RALT:          /* right alt */
            // strip out the Shift-Control-Alt keys so the user doesn't see them
            break;

        default:
        case INSERT_KEY:        /* insert key */
        case UARROW_KEY:
        case DARROW_KEY:
        case LARROW_KEY:
        case RARROW_KEY:
        case F1K:
        case F2K:
        case F3K:
        case F4K:
        case F5K:
        case F6K:
        case F7K:
        case F8K:
        case F9K:
        case F10K:
        case F11K:
        case F12K:
            // if a callback has been registered
            if (Keyboard_Callback != NULL){
                // invoke it with the key just pressed
                (*Keyboard_Callback)(keyPress, ControlActive, AltActive);
            }
```

```
            break;
    }
}

// update our state machine with the scan code just received
static void ProcessScanCode(unsigned char scanCode){

    unsigned char key;

    switch (scanCode){

        case ExtendedScanCode:
            KbdByteState = EXTENDEDMAKE;
            break;

        default:
            switch (KbdByteState){

                // waiting for 1st byte of packet to determine packet type
                case SINGLEMAKE:
                    // if BreakScanCode received (while not in extended mode)
                    if (scanCode == BreakScanCode){
                        // setup to process next character as a SINGLEMAKE character being released
                        KbdByteState = BREAKCODE;
                    }else{
                        // process a SINGLEMAKE being pressed
                        if (scanCode < NUM_ENTRIES(ScanCodes)){

                            // Get the un-shifted key code
                            key = ScanCodes[scanCode];
                            if ((key >= 'a') && (key <= 'z')){
                                // if a 'letter', it will be UPPERCASE if caps lock active OR shift active BUT NOT BOTH
                                if (CapsLockActive ^ ShiftActive){
                                    key = ShiftedScanCodes[scanCode];
                                }
                            }else{
                                // all other keys are shifted if shift active
                                if (ShiftActive){
                                    key = ShiftedScanCodes[scanCode];
                                }
                            }

                            if (key != UIMP){

                                switch (key){

                                    case CPLK:
                                        // caps lock pressed, toggle the caps lock state
                                        CapsLockActive = !CapsLockActive;
                                        // update keyboard LED states
                                        Keyboard_Set_LEDs(NumLockActive, CapsLockActive, ScrollLockActive);
                                        break;
                                    case LSHT:
                                    case RSHT:
                                        // left or right shift pressed, indicate shift is active
                                        ShiftActive = true;
                                        break;
                                    case LALT:
                                    case RALT:
                                        // left or right alt pressed, indicate alt is active
                                        AltActive = true;
                                        break;
                                    case LCTL:
                                    case RCTL:
                                        // left or right control pressed, indicate control is active
```

```
                            ControlActive = true;
                            break;
                        default:
                            // just a normal key, inform the caller that a key press occurred
                            HandleKeyPress(key);
                            break;
                    }
                }
            }
        }
        break;

    // ExtendedScanCode rxd, waiting for 2nd byte to determine if press or release
    case EXTENDEDMAKE:
        // if BreakScanCode received while in extended mode
        if (scanCode == BreakScanCode){
            // setup to receive a 3rd byte specifying which extended character was just released
            KbdByteState = EXTENDEDBREAK;
        }else{
            // handle an extended character was just pressed
            switch (scanCode){
                case 0x4a:
                    HandleKeyPress(ShiftActive ? '?' : '/');
                    break;
                case 0x7a:
                    HandleKeyPress(0x7f);   // DEL key
                    break;
                case 0x14:
                    HandleKeyPress(RCTL);
                    break;
                case 0x11:
                    HandleKeyPress(RALT);
                    break;
                case 0x70:
                    HandleKeyPress(INSERT_KEY);
                    break;
                case 0x75:
                    HandleKeyPress(UARROW_KEY);
                    break;
                case 0x72:
                    HandleKeyPress(DARROW_KEY);
                    break;
                case 0x6b:
                    HandleKeyPress(LARROW_KEY);
                    break;
                case 0x74:
                    HandleKeyPress(RARROW_KEY);
                    break;
            }
            // sequence is complete
            KbdByteState = SINGLEMAKE;
        }
        break;

    // BreakScanCode rxd (without ExtendedScanCode), waiting for 2nd byte to determine key released
    case BREAKCODE:
        // only process scan codes in our table
        if (scanCode < NUM_ENTRIES(ScanCodes)){
            key = ScanCodes[scanCode];

            // ignore unimplemented scan codes
            if (key != UIMP){

                // handle 'Shift' 'Alt' and 'Control' keys being released
```

```
                        // handle "shift", "alt" and "control" keys being released
                        // ignore all other keys being released
                        switch (key){
                            case LSHT:
                            case RSHT:
                                // left or right shift released, indicate shift is inactive
                                ShiftActive = false;
                                break;
                            case LALT:
                            case RALT:
                                // left or right alt released, indicate alt is inactive
                                AltActive = false;
                                break;
                            case LCTL:
                            case RCTL:
                                // left or right control released, indicate control is inactive
                                ControlActive = false;
                                break;
                            default:
                                break;
                        }
                    }
                }

                KbdByteState = SINGLEMAKE;
                break;

            // ExtendedScanCode followed by BreakScanCode rxd, waiting for 3rd byte to determine extended key released
            case EXTENDEDBREAK:
                // ignore extended scan codes being released
                KbdByteState = SINGLEMAKE;
                break;

            // error condition
            default:
                KbdByteState = SINGLEMAKE;
                break;
        }
        break;
    }
}
```

```
/*
 * 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 "Keyboard.h"
#include "Timer.h"
#include "RegisterIF.h"
#include "ShiftOut.h"



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

    // invoke other interrupt modules
    Keyboard_IOC_Handler();
    Timer_Handler();
}



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

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

```
    ANSELC = 0x00;

    // Initialize all the modules
    Timer_Init();
    ShiftOut_Init();
    Keyboard_Init();
    RegisterIF_Init();

    ei(); // enable all interrupts

    while (1){

        // 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 "Keyboard.h"
#include "Timer.h"

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

void Process_Key_Press(unsigned char keypressed, bool ControlActive, bool AltActive);
void RegisterIF_Handle_Request(void);

unsigned char Key_Pressed;
bool Control_Active;
bool Alt_Active;

unsigned char Last_ShiftOut_Value = 0xff;

//*****************************
// 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
    Key_Pressed = KYBD_NOKEY_PRESSED;
    Control_Active = false;
    Alt_Active = false;

    // register a callback to handle key presses
    Keyboard_Register_Callback(Process_Key_Press);
}

// 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 keyboard status byte
            // Note: Num lock, Scroll lock, and Shift currently not implemented
            ShiftOutValues[0] = (KeyboardDetected ? KYBD_FOUND : 0x00)
                    | (HardwareFailure ? KYBD_HARDERR : 0x00)
                    | (CapsLockActive ? KYBD_CAPS_LOCK : 0x00)
                    | (Control_Active ? KYBD_CONTROL_ACTIVE : 0x00)
                    | (Alt_Active ? KYBD_ALT_ACTIVE : 0x00);
            break;
        case INPUT_RD_ADDR1:                    // Read last key pressed
            ShiftOutValues[0] = Key_Pressed;
            Key_Pressed = KYBD_NOKEY_PRESSED;
            break;
        case INPUT_RD_ADDR2:                    // Read free running 7.936mS Tick Counter
            ShiftOutValues[0] = TickCounter;
            break;

        case INPUT_RD_ADDR3:                            // Read Expansion ID register
            ShiftOutValues[0] = IF_FEATURE_KEYBOARD;    // indicate this board implements the keyboard feature
            break;

        // No write commands currently implemented on the keyboard IF
        case INPUT_WR_ADDR0:
            WriteRegs[0] = ShiftInValues[0];
            break;
        case INPUT_WR_ADDR1:
            WriteRegs[1] = ShiftInValues[0];
```

```
            WriteRegs[1] = ShiftInValues[0];
            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;
}

// Key pressed callback handler
```

```
// Key pressed callback handler
void Process_Key_Press(unsigned char keypressed, bool ControlActive, bool AltActive){

    Key_Pressed = keypressed;
    Control_Active = ControlActive;
    Alt_Active = AltActive;
}
```

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

// 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 */

// 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;
    }
}
```

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

unsigned char TickCounter;

void Timer0_Init(void);
void Timer0_Handler(void);
void Timer2_Init(void);
void Timer2_Handler(void);

// Timer Initialization
void Timer_Init(void){

    Timer0_Init();
    Timer2_Init();
}

// Timer Initialization
void Timer_Handler(void){

    Timer0_Handler();
    Timer2_Handler();
}

// free running timer
void Timer0_Init(void){

    // Initialize Timer0 to interrupt every 2mS
    // 31.25KHz / 1 (postscaler) / 248 (TMR0H) = 126Hz (7.936mS)
    // REMEMBER to update Timer0uSPeriod (timer.h) 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 = 4;          // use LFINTOSC as the clock source
    TMR0H = 248;                // count up to 248
    T0CON0bits.EN = 1;          // enable timer
    PIE0bits.TMR0IE = 1;        // enable interrupts for timer 0
    PIR0bits.TMR0IF = 0;        // clear any pending interrupt condition


}

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

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

       // increment the tick counter
       TickCounter++;

       // clear this interrupt condition
       PIR0bits.TMR0IF = 0;
   }
}
```

```
// Timer 2 initialization
void Timer2_Init(void){

    T2CONbits.ON = 0;       // disabled
    T2CLKCONbits.CS = 6;    // use MFINTOSC (31.25KHz) clock source
    T2HLTbits.MODE = 8;     // configure as one-shot, software start
    T2HLTbits.PSYNC = 0;    // sync prescaler output to Fosc/4
    T2HLTbits.CKSYNC = 0;   // sync ON bit to timer clock input
    T2TMR = 0;
    T2CONbits.CKPS = 3;     // 1:8 prescaler
    T2CONbits.OUTPS = 0x0;  // 1:1 postscaler

    // Period = 31.25KHz / 8 / 1 = 3.906KHz (256uS)
    // Remember to change Timer2uSPeriod if you change this

    TMR2IE = 1;         // enable interrupts for T2
    PEIE = 1;           // ensure that PEIE is set (not needed for Timer0, required for Timer1/2)
}

// Valid range for uSeconds is: 256uS (Timer2uSPeriod) - 65,280uS (Timer2uSPeriod * 255)
// (values outside this range will be clamped to the range limits)
void Start_Timer2(unsigned int uSeconds){

    T2CONbits.ON = 0;                                       // disable the timer while it is reconfigured


    // scale the requested value to the supported range
    if (uSeconds < Timer2uSPeriod) uSeconds = Timer2uSPeriod;
    if (uSeconds > ((unsigned long) Timer2uSPeriod * 255)) uSeconds = ((unsigned int) Timer2uSPeriod * 255);

    T2TMR = 0;                                              // reset the counter
    T2PR = (unsigned char) (uSeconds / Timer2uSPeriod);     // setup the requested timeout value
    TMR2IF = 0;                                             // clear any prior T2 overflow flag
    T2CONbits.ON = 1;                                       // enable the timer and wait for the interrupt
}

// disable the timeout if it is no longer needed
void Stop_Timer2(void){

    T2CONbits.ON = 0;                                       // disable the timer
}

// interrupt handler
void Timer2_Handler(void){

    if (TMR2IE && TMR2IF){

        TMR2IF = 0;

        Keyboard_Timer_Handler();
    }
}
```