```c
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "encoderFunctions.h"
#include "LabFA.h"

//#define TRUE 1
//#define FALSE 0
//#define true 1
//#define false 0
//#define True 1
//#define False 0

// Global variables from other files
//extern uint8_t segment_data[5];
//extern uint8_t dec_to_7seg[12];
//extern uint8_t digitSelect[8];
//extern volatile uint8_t buttonState;
// Number displayed to 7seg
//extern volatile uint16_t segNum;
// Number displayed to bargraph
//extern volatile uint8_t barNum;


// Holds state of encoders
volatile uint8_t encoderState;
extern uint16_t current_fm_freq;
extern volatile uint8_t freqTime;
extern volatile uint8_t needToChangeStation;



//******************************************************************************
//       -- Performs Logic to Test Direction of Encoder Movement --
//******************************************************************************
void interpret_encoders(){
    uint8_t curr=0;
    uint8_t prev=0;
    volatile static uint8_t encR_cwse = 0;
    volatile static uint8_t encR_ccws = 0;
    volatile static uint8_t encL_cwse = 0;
    volatile static uint8_t encL_ccws = 0;
    volatile static uint8_t encStatusReg=0;

    // encStatusReg variable decoding
    // bit7     bit6    bit5    bit4    bit3    bit2    bit1    bit0
    //                  LWFN    RWFN    LPrv    LPrv    RPrv    RPrv
    // WFN = Wait for Next

    // Encoder states
    curr = (encoderState & 0x0F);
    prev = (encStatusReg & 0x0F);

    // Right Encoder Changed State
    if ((curr & RMSK) != (prev & RMSK)) {

        // Shift registers to keep track of turning speed
        switch(checkDirection((curr & RMSK),(prev & RMSK))) {
            case 0b01:
                encR_cwse = (encR_cwse<<1)|1;
                encR_ccws =  encR_ccws>>1;
```

```c
                break;
            case 0b10:
                encR_cwse =  encR_cwse>>1;
                encR_ccws = (encR_ccws<<1)|1;
                break;
            default:
                encR_cwse = encR_cwse>>1;
                encR_ccws = encR_ccws>>1;
                break;
        }

        // When at notch, reset turning speed
        if ((curr & RMSK) == RMSK) {
            encR_cwse = 0;
            encR_ccws = 0;
        }

        // Check right encoder
        if (encStatusReg & (1<<RWFN)) {

            if (encR_cwse >= 0b11) {

                // Extra increments to compensate for missed bits
                if (encR_cwse >= 0b11111) {
                    if (encR_cwse >= 0b111111) {
                        increment(RIGHT);
                    }
                    increment(RIGHT);
                }

                increment(RIGHT);
                encStatusReg &= ~(1<<RWFN);
                encR_cwse = 0;
                encR_ccws = 0;

            } else if (encR_ccws >= 0b11) {

                // Extra decrements to compensate for missed bits
                if (encR_ccws >= 0b11111) {
                    if (encR_ccws >= 0b111111) {
                        decrement(RIGHT);
                    }
                    decrement(RIGHT);
                }

                decrement(RIGHT);
                encStatusReg &= ~(1<<RWFN);
                encR_cwse = 0;
                encR_ccws = 0;
            }
        }

        // When at halfway point, enable state change
        //  This prevents a floating state next to notch triggering an event
        if ((curr & RMSK) == 0x00) {
            encStatusReg |= (1<<RWFN);
        }

        encStatusReg &= ~RMSK;
        encStatusReg |= (encoderState & RMSK);
    }

    // Left Encoder Changed State
    if ((curr & LMSK) != (prev & LMSK)) {

        // Shift registers to keep track of turning speed
        switch(checkDirection(((curr & LMSK)>>2),((prev & LMSK)>>2))) {
            case 0b01:
                encL_cwse = (encL_cwse<<1)|1;
                encL_ccws =  encL_ccws>>1;
                break;
            case 0b10:
                encL_cwse =  encL_cwse>>1;
                encL_ccws = (encL_ccws<<1)|1;
                break;
            default:
                encL_cwse = encL_cwse>>1;
                encL_ccws = encL_ccws>>1;
                break;
        }

        // When at notch, reset turning speed
        if ((curr & LMSK) == LMSK) {
            encL_cwse = 0;
            encL_ccws = 0;
        }

        // Check right encoder
        if (encStatusReg & (1<<LWFN)) {

            if (encL_cwse >= 0b1) {

                // Extra increments to compensate for missed bits
                if (encL_cwse >= 0b11111) {
                    if (encL_cwse >= 0b111111) {
                        increment(LEFT);
                    }
                    increment(LEFT);
                }

                increment(LEFT);
                encStatusReg &= ~(1<<LWFN);
                encL_cwse = 0;
                encL_ccws = 0;

            } else if (encL_ccws >= 0b1) {

                // Extra decrements to compensate for missed bits
                if (encL_ccws >= 0b11111) {
                    if (encL_ccws >= 0b111111) {
                        decrement(LEFT);
                    }
                    decrement(LEFT);
                }

                decrement(LEFT);
                encStatusReg &= ~(1<<LWFN);
                encL_cwse = 0;
                encL_ccws = 0;
            }
        }
```

```c
        // When at halfway point, enable state change
        //  This prevents a floating state next to notch triggering an event
        if ((curr & LMSK) == 0x00) {
            encStatusReg |= (1<<LWFN);
        }

        encStatusReg &= ~LMSK;
        encStatusReg |= (encoderState & LMSK);
    }
}




//******************************************************************************
//    -- Encoder Checker
//    Return Value
//    bit1    bit0
//    0       1   Clockwise
//    1       0   Counter Clockwise
//******************************************************************************
uint8_t checkDirection(uint8_t currLocal, uint8_t prev_local) {

    currLocal &= 0b11;
    prev_local &= 0b11;
    switch (currLocal) {
        case 0b01:
            switch (prev_local){
                case 0b11:
                    return CWSE;
                case 0b00:
                    return CCWS;
            }
            break;
        case 0b00:
            switch (prev_local){
                case 0b01:
                    return CWSE;
                case 0b10:
                    return CCWS;
            }
            break;
        case 0b10:
            switch (prev_local){
                case 0b00:
                    return CWSE;
                case 0b11:
                    return CCWS;
            }
            break;
        case 0b11:
            switch (prev_local){
                case 0b10:
                    return CWSE;
                case 0b01:
                    return CCWS;
            }
            break;
    }//switch

    return 0;
}
```

```
//extern enum states {SETs_TIME};
//extern enum states {DISP_TIME, SET_TIME, ALARM, SNOOZE, SET_ALARM};
//extern enum states STATE;
extern enum states STATE;
extern uint8_t clock_m;
extern uint8_t clock_h;
extern uint8_t alarm_m;
extern uint8_t alarm_h;
extern uint8_t volume;
//******************************************************************************
//    -- Conditionally Increment Based on State
//******************************************************************************
void increment(uint8_t LR) {
    switch (STATE) {
        case SET_TIME:
            if (LR == LEFT) {
                clock_h++;
            } else {
                clock_m++;
            }
            break;
        case SET_ALARM:
            if (LR == LEFT) {
                alarm_h++;
            } else {
                alarm_m++;
            }
            break;
        case DISP_TIME:
            if (LR == LEFT) {
                                freqTime = 0;
                                needToChangeStation = 1;
                    current_fm_freq += 20;
                                if (current_fm_freq >= 10810) {
                                        needToChangeStation = 0;
                                        current_fm_freq = 10810;
                                }
                        } else {
                                volume += 10;
                                if (volume < 10)
                                        volume = 250;
                        }
            break;
        default:
            break;
    }
}



//******************************************************************************
//    -- Conditionally Decrement Based on State
//******************************************************************************
void decrement(uint8_t LR) {
    switch (STATE) {
        case SET_TIME:
            if (LR == LEFT) {
                clock_h--;
```

```
            } else {
                clock_m--;
            }
            break;
        case SET_ALARM:
            if (LR == LEFT) {
                alarm_h--;
            } else {
                alarm_m--;
            }
            break;
        case DISP_TIME:
                        if (LR == LEFT) {
                                freqTime = 0;
                                needToChangeStation = 1;
                                current_fm_freq -= 20;
                                if (current_fm_freq <= 8790) {
                                        needToChangeStation = 0;
                                        current_fm_freq = 8790;
                                }
            } else {
                                volume -= 10;
                                if (volume > 240)
                                        volume = 0;
                        }
        default:
            break;
    }
}
```