ELEC3042 Minor Project Report

Name: Joe Sample

SID: 10101010

Overview

In implementing the home alarm system, I broke down the system into three components: an input handler, main system, and four output generators (one for each of the LEDs and one for the siren). The input handler manages external inputs and sets the appropriate internal signals. The main system processes the signals from the input handler, updates the system state and generates appropriate outputs depending on what state the system is in. Figure 1 shows the system block diagram and the signal flows between them.

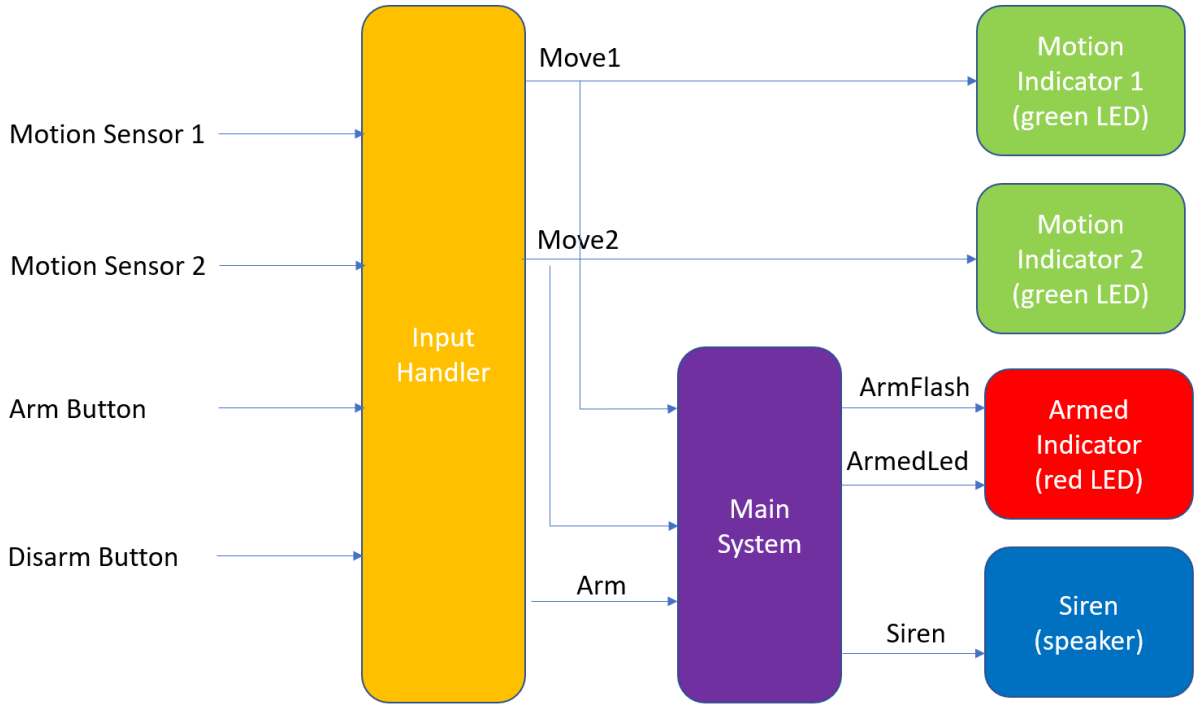

Figure 1 System Block Diagram showing inputs, outputs and signal flows

System Resources

To implement the system, I used Timer0 to keep time, and Timer1 for generating the siren sweep. Timer1 was chosen for generating the siren sweep because it provides better frequency resolution. Timer0 was set up in CTC mode to generate an interrupt every 1 ms. Timer1 was also set up in CTC mode to generate the alarm siren sound on pin 9. Changes in the inputs (motion sensors, arm and disarm button) generated an interrupt via PCINT2. The following code shows how these resources were set up.

```
void timer1setup() {
    // Timer1 for siren
    TCCR1A = 0b01000000;    // CTC mode
    TCCR1B = 0b00001011;    // /64 prescaler
    TCCR1C = 0;
    TCNT1 = 0;
    OCR1A = 0;
    OCR1B = 0;
    ICR1 = 0;
    TIMSK1 = 0;
    TIFR1 = 0;
}

void setup() {
    // use PORTD for inputs (Arm, Clear, Sensor1, Sensor2: PD2 to PD5)
    DDRD &= 0b11000011;
    PORTD |= 0b00111100;

    // use PORTB for outputs (Armed, Siren, Sensor1, Sensor2: PB0 to PB3)
    DDRB |= 0b00001111;
    PORTB &= 0b11110000;

    // Timer0 for system timing - interrupt every 1 ms
    TCCR0A = 0b00000010;    // CTC mode
    TCCR0B = 0b00000011;    // /64 prescaler
    TCNT0 = 0;
    OCR0A = 250;            // value for 1 ms
    OCR0B = 0;
    TIMSK0 = 0b00000010;    // enable interrupt on compare match A
    TIFR0 = 0;

    // set up pin change interrupt for PORTD
    EICRA = 0;
    EIMSK = 0;
    EIFR = 0;
    PCICR = 0b00000100;     // enable PCIE2
    PCIFR = 0;
    PCMSK2 = 0b00111100;    // enable interrupts of input pins
    PCMSK1 = 0;
    PCMSK0 = 0;

    sei();
}
```

Input Handler

The input handler is implemented in the process_inputs() function. This function examines the status of PORTD at the time the pin change interrupt was generated and sets the corresponding internal signals to the appropriate level.

```
void process_inputs() {
    if ((inputs & ARMBUTTON) == 0) { // Arm button pressed
        flags |= ARMF;
    }

    if ((inputs & CLEARBUTTON) == 0) { // clear button pressed
        flags &= ~ARMF;
    }

    if ((inputs & SENSOR1) == 0) { // sensor 1 triggered
        flags |= MOVE1F;
    } else {
        flags &= ~MOVE1F;
    }

    if ((inputs & SENSOR2) == 0) { // sensor 2 triggered
        flags |= MOVE2F;
    } else {
        flags &= ~MOVE2F;
    }
}
```

Main System State Diagram

Figure 2 shows the state diagram of the main system and the corresponding state transition table is shown as Table 1.

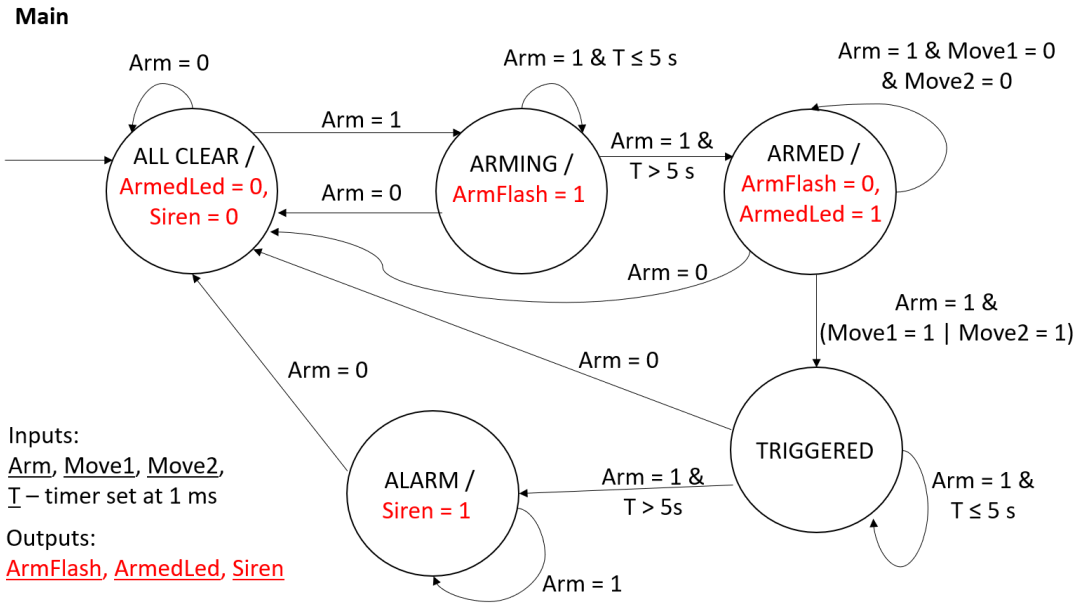

Figure 2 Main System State Diagram. Arm, Move1, Move2 are signals generated by the input handler. T is a variable that keeps track of time. ArmFlash, ArmedLed and Siren are signals generated

Table 1 Main System State Transition Table

Inputs				Current State	Next State	Outputs		
T	Move1	Move2	Arm			ArmedLed	ArmFlash	Siren
X	X	X	0	All Clear	All Clear	0	0	0
X	X	X	1	All Clear	Arming	0	0	0
X	X	X	0	Arming	All Clear	0	1	0
< 5 s	X	X	1	Arming	Arming	0	1	0
> 5 s	X	X	1	Arming	Armed	0	1	0
X	X	X	0	Armed	All Clear	1	0	0
X	0	0	1	Armed	Armed	1	0	0
X	X	1	1	Armed	Triggered	1	0	0
X	1	X	1	Armed	Triggered	1	0	0
X	X	X	0	Triggered	All Clear	1	0	0
< 5 s	X	X	1	Triggered	Triggered	1	0	0
> 5 s	X	X	1	Triggered	Alarm	1	0	0
X	X	X	0	Alarm	All Clear	1	0	1
X	X	X	1	Alarm	Alarm	1	0	1

The main system state diagram is implemented within the main() function. The code corresponding section of code is as follows:

```
/** update system state **/
switch (cur_state) {
    case CLEAR:
        flags &= 0b11100011;    // no siren or red LED
        if ((flags & ARMF) != 0) {
            T = cur_ms + 5000;
            cur_state = ARMING;
        }
        break;
    case ARMING:
        flags |= ARMFLASHF;     // set ArmFlash
        if ((flags & ARMF) == 0) {
            cur_state = CLEAR;
        } else if (((flags & ARMF) != 0) && (cur_ms > T)) {
            flags &= ~ARMFLASHF;    // clear ArmFlash
            cur_state = ARMED;
        }
        break;
    case ARMED:
        flags |= ARMEDLEDF;     // set ArmedLED
        if (((flags & ARMF) != 0) &&
                (((flags & MOVE1F) != 0) || ((flags & MOVE2F) != 0))) {
            T = cur_ms + 5000;
            cur_state = TRIGGERED;
        } else if ((flags & ARMF) == 0) {
            cur_state = CLEAR;
        }
        break;
    case TRIGGERED:
        if (((flags & ARMF) != 0) && (cur_ms > T)) {
            cur_state = ALARM;
        } else if ((flags & ARMF) == 0) {
            cur_state = CLEAR;
        }
        break;
    case ALARM:
        flags |= SIRENF;
        if ((flags & ARMF) == 0) {
            cur_state = CLEAR;
        }
        break;
}
```

Motion LEDs

The code for turning on and off the LEDs whenever motion is detected/not detected is shown below. It simply checks whether the movement-related bits in the flag register has been set or not.

```
void sensor1_led() {
    if ((flags & MOVE1F) != 0) {
        PORTB |= _BV(PORTB2);
    } else {
        PORTB &= ~_BV(PORTB2);
    }
}

void sensor2_led() {
    if ((flags & MOVE2F) != 0) {
        PORTB |= _BV(PORTB3);
    } else {
        PORTB &= ~_BV(PORTB3);
    }
}
```

Red Armed LED

The state diagram describing the output on the Red Armed LED is shown in Fig. 3 and the corresponding state transition table is shown in Table 2. This state diagram allows me to implement both flashing and always on functionality with a smaller number of states.

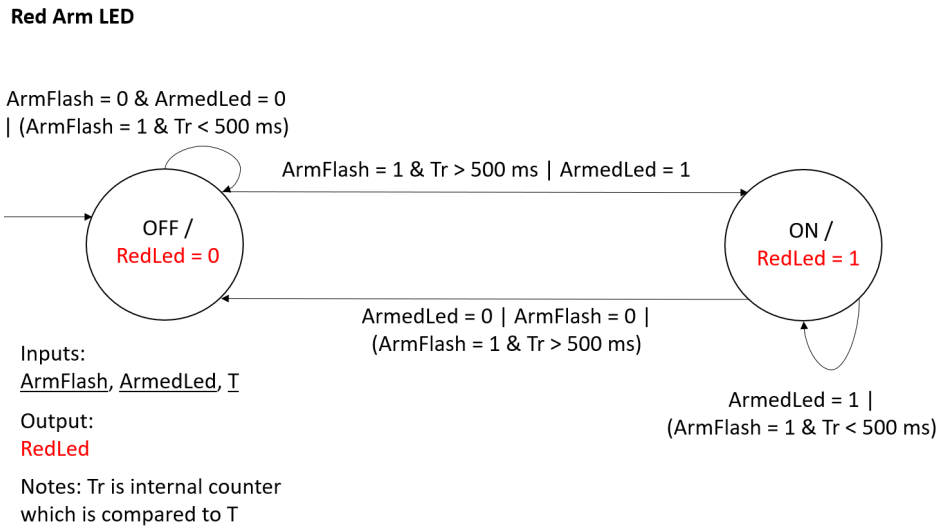

Figure 3 State diagram for producing flashing and always on functionality of the Red Armed LED.

Table 2 State Transition Table for Red Armed LED

Inputs					Output
Tr	ArmLed	ArmFlash	Current State	Next State	RedLed
X	0	0	OFF	OFF	0
< 500 ms	0	1	OFF	OFF	0
> 500 ms	X	1	OFF	ON	0
X	1	X	OFF	ON	0
X	0	0	ON	OFF	1
< 500 ms	0	1	ON	ON	1
> 500 ms	X	1	ON	OFF	1
X	1	X	ON	ON	1

The state diagram shown in Fig.3 is implemented in the armed_led() function and is shown below.

```
enum REDSTATE {OFF, ON};
enum REDSTATE redled_state = OFF;
uint32_t Tr = 0; // keeps track of time passed for red LED

void armed_led(uint32_t cur_ms) {
    switch (redled_state) {
        case OFF:
            PORTB &= ~_BV(PORTB0);
            if ((((flags & ARMFLASHF) == 0) && ((flags & ARMEDLEDF) == 0)) ||
                    (((flags & ARMFLASHF) != 0) && cur_ms <= Tr)) {
                redled_state = OFF;
            } else if (((flags & ARMFLASHF) != 0) || ((flags & ARMEDLEDF) != 0)) {
                Tr = cur_ms + 500;
                redled_state = ON;
            }
            break;
        case ON:
            PORTB |= _BV(PORTB0);
            if (((flags & ARMEDLEDF) != 0) || (((flags & ARMFLASHF) != 0) && cur_ms <= Tr))  {
                redled_state = ON;
            } else if ( (((flags & ARMFLASHF) != 0) && cur_ms > Tr) ||
                    ((flags & ARMEDLEDF) == 0) || ((flags & ARMFLASHF) == 0) ) {
                Tr = cur_ms + 500;
                redled_state = OFF;
            }
            break;
        default:
            redled_state = OFF;
    }
}
```

Siren

The state diagram for producing the alarm siren sound is shown in Fig. 4. The siren sweeps through five frequencies within 2 s by updating the OCR1A value every 400 ms. If the system is disarmed during the sweep, it will turn off the siren before the sweep ends.

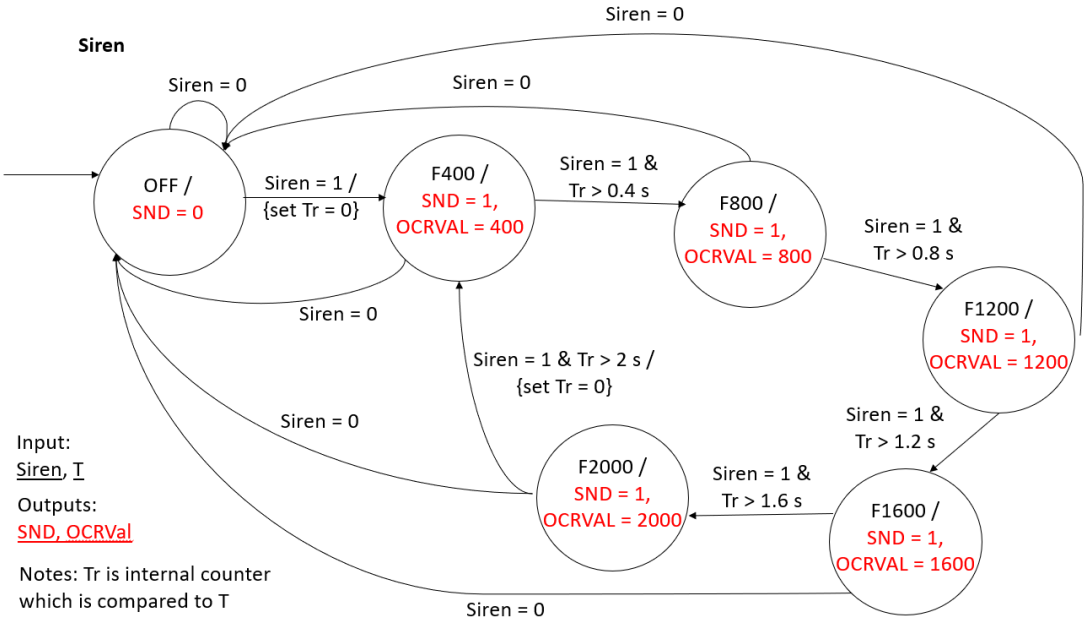

Figure 4 State diagram for producing the siren sound.

This state diagram is implemented in the siren() function.

```
enum SIRENSTATE {FOFF, F400, F800, F1200, F1600, F2000};
enum SIRENSTATE cur_siren_state = FOFF;
uint32_t Ts = 0; // keeps track of time passed for red LED

void siren(uint32_t cur_ms) {
    switch (cur_siren_state) {
        case FOFF:
            if ((flags & SIRENF) != 0) {
                Ts = cur_ms + 400;
                cur_siren_state = F400;
                timer1setup();
            } else {
                TCCR1A = 0;
                TCCR1B = 0;
            }
            break;
        case F400:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F800;
                TCNT1 = 0;
                OCR1A = 156;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F800:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F1200;
                TCNT1 = 0;
                OCR1A = 104;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F1200:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F1600;
                TCNT1 = 0;
                OCR1A = 78;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F1600:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F2000;
                TCNT1 = 0;
                OCR1A = 63;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F2000:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F400;
                TCNT1 = 0;
                OCR1A = 313;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        default:
            cur_siren_state = FOFF;
    }
}
```

Overview of main loop code

At each iteration of the main system loop, the number of milliseconds since the start of the system is read. The number of milliseconds is used for keeping track of how long the system has been in particular states, and also for generating the flashing LED and siren outputs. The state of the system inputs is then checked which, in turn, updates the appropriate internal signals (stored in an 8-bit register called flags) for input changes. The bits in the flags register are assigned to: [Move1, Move2, Arm, ArmFlash, ArmedLED, Siren, unused, unused]. These names correspond to the internal system signals shown in Fig. 1.

Following this, the system state is updated in response to changes in the input. Lastly, the outputs are generated depending on the current system state. The full code for the system is provided on the following pages.

```c
/*
 * File:   home_alarm.c
 * Author: Alan
 *
 * Created on 8 March 2022, 10:01 AM
 */


#include <avr/io.h>
#include <avr/interrupt.h>

#define ARMBUTTON   0b00000100
#define CLEARBUTTON 0b00001000
#define SENSOR1     0b00010000
#define SENSOR2     0b00100000

#define MOVE1F      0b10000000
#define MOVE2F      0b01000000
#define ARMF        0b00100000
#define ARMFLASHF   0b00010000
#define ARMEDLEDF   0b00001000
#define SIRENF      0b00000100

volatile uint8_t inputs = 0b00111100;
volatile uint32_t clock_count = 0;

uint8_t flags = 0; // internal signals
// [Move1, Move2, Arm, ArmFlash, ArmedLED, Siren, unused, unused]

uint32_t millis() {
    /*
     * Return the current clock_count value.
     * We temporarily disable interrupts to ensure the clock_count value
     * doesn't change while we are reading them.
     *
     * We then restore the original SREG, which contains the I flag.
     */
    register uint32_t count;
    register char cSREG;

    cSREG = SREG;
    cli();
    count = clock_count;
    SREG = cSREG;
    return count;
}

ISR(PCINT2_vect) {
    inputs = PIND;
}

ISR(TIMER0_COMPA_vect) {
    clock_count++;
}

void timer1setup() {
    // Timer1 for siren
    TCCR1A = 0b01000000;    // CTC mode
    TCCR1B = 0b00001011;    // /64 prescaler
    TCCR1C = 0;
    TCNT1 = 0;
    OCR1A = 0;
    OCR1B = 0;
    ICR1 = 0;
    TIMSK1 = 0;
    TIFR1 = 0;
}

void setup() {
    // use PORTD for inputs (Arm, Clear, Sensor1, Sensor2: PD2 to PD5)
    DDRD &= 0b11000011;
    PORTD |= 0b00111100;

```

```
74      // use PORTB for outputs (Armed, Siren, Sensor1, Sensor2: PB0 to PB3)
75      DDRB |= 0b00001111;
76      PORTB &= 0b11110000;
77
78      // Timer0 for system timing - interrupt every 1 ms
79      TCCR0A = 0b00000010;    // CTC mode
80      TCCR0B = 0b00000011;    // /64 prescaler
81      TCNT0 = 0;
82      OCR0A = 250;            // value for 1 ms
83      OCR0B = 0;
84      TIMSK0 = 0b00000010;    // enable interrupt on compare match A
85      TIFR0 = 0;
86
87      // set up pin change interrupt for PORTD
88      EICRA = 0;
89      EIMSK = 0;
90      EIFR = 0;
91      PCICR = 0b00000100;     // enable PCIE2
92      PCIFR = 0;
93      PCMSK2 = 0b00111100;    // enable interrupts of input pins
94      PCMSK1 = 0;
95      PCMSK0 = 0;
96
97      sei();
98  }
99
100 void process_inputs() {
101     if ((inputs & ARMBUTTON) == 0) { // Arm button pressed
102         flags |= ARMF;
103     }
104
105     if ((inputs & CLEARBUTTON) == 0) { // clear button pressed
106         flags &= ~ARMF;
107     }
108
109     if ((inputs & SENSOR1) == 0) { // sensor 1 triggered
110         flags |= MOVE1F;
111     } else {
112         flags &= ~MOVE1F;
113     }
114
115     if ((inputs & SENSOR2) == 0) { // sensor 2 triggered
116         flags |= MOVE2F;
117     } else {
118         flags &= ~MOVE2F;
119     }
120 }
121
122 void sensor1_led() {
123     if ((flags & MOVE1F) != 0) {
124         PORTB |= _BV(PORTB2);
125     } else {
126         PORTB &= ~_BV(PORTB2);
127     }
128 }
129
130 void sensor2_led() {
131     if ((flags & MOVE2F) != 0) {
132         PORTB |= _BV(PORTB3);
133     } else {
134         PORTB &= ~_BV(PORTB3);
135     }
136 }
137
138 enum REDSTATE {OFF, ON};
139 enum REDSTATE redled_state = OFF;
140 uint32_t Tr = 0; // keeps track of time passed for red LED
141
142 void armed_led(uint32_t cur_ms) {
143     switch (redled_state) {
144         case OFF:
145             PORTB &= ~_BV(PORTB0);
146             if ((((flags & ARMFLASHF) == 0) && ((flags & ARMEDLEDF) == 0)) ||
```

```
                    (((flags & ARMFLASHF) != 0) && cur_ms <= Tr)) {
                redled_state = OFF;
            } else if (((flags & ARMFLASHF) != 0) || ((flags & ARMEDLEDF) != 0)) {
                Tr = cur_ms + 500;
                redled_state = ON;
            }
            break;
        case ON:
            PORTB |= _BV(PORTB0);
            if (((flags & ARMEDLEDF) != 0) || (((flags & ARMFLASHF) != 0) && cur_ms
            <= Tr))  {
                redled_state = ON;
            } else if ( (((flags & ARMFLASHF) != 0) && cur_ms > Tr) ||
                    ((flags & ARMEDLEDF) == 0) || ((flags & ARMFLASHF) == 0) ) {
                Tr = cur_ms + 500;
                redled_state = OFF;
            }
            break;
        default:
            redled_state = OFF;
    }
}

enum SIRENSTATE {FOFF, F400, F800, F1200, F1600, F2000};
enum SIRENSTATE cur_siren_state = FOFF;
uint32_t Ts = 0; // keeps track of time passed for red LED

void siren(uint32_t cur_ms) {
    switch (cur_siren_state) {
        case FOFF:
            if ((flags & SIRENF) != 0) {
                Ts = cur_ms + 400;
                cur_siren_state = F400;
                timer1setup();
            } else {
                TCCR1A = 0;
                TCCR1B = 0;
            }
            break;
        case F400:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F800;
                TCNT1 = 0;
                OCR1A = 156;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F800:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F1200;
                TCNT1 = 0;
                OCR1A = 104;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F1200:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F1600;
                TCNT1 = 0;
                OCR1A = 78;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F1600:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F2000;
```

```
                TCNT1 = 0;
                OCR1A = 63;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        case F2000:
            if (((flags & SIRENF) != 0) && (cur_ms > Ts)) {
                Ts = cur_ms + 400;
                cur_siren_state = F400;
                TCNT1 = 0;
                OCR1A = 313;
            } else if ((flags & SIRENF) == 0) {
                cur_siren_state = FOFF;
            }
            break;
        default:
            cur_siren_state = FOFF;
    }
}

enum STATE {CLEAR, ARMING, ARMED, TRIGGERED, ALARM};

int main(void) {
    uint32_t T = 0;                 // keeps track of time passed for system states
    uint32_t cur_ms = 0;            // current clock count
    enum STATE cur_state = CLEAR;   // current system state

    setup();
    while (1) {
        /** get current clock count*/
        cur_ms = millis();
        sei();

        /** handle inputs **/
        process_inputs();

        /** update system state **/
        switch (cur_state) {
            case CLEAR:
                flags &= 0b11100011;    // no siren or red LED
                if ((flags & ARMF) != 0) {
                    T = cur_ms + 5000;
                    cur_state = ARMING;
                }
                break;
            case ARMING:
                flags |= ARMFLASHF;     // set ArmFlash
                if ((flags & ARMF) == 0) {
                    cur_state = CLEAR;
                } else if (((flags & ARMF) != 0) && (cur_ms > T)) {
                    flags &= ~ARMFLASHF;    // clear ArmFlash
                    cur_state = ARMED;
                }
                break;
            case ARMED:
                flags |= ARMEDLEDF;     // set ArmedLED
                if (((flags & ARMF) != 0) &&
                        (((flags & MOVE1F) != 0) || ((flags & MOVE2F) != 0))) {
                    T = cur_ms + 5000;
                    cur_state = TRIGGERED;
                } else if ((flags & ARMF) == 0) {
                    cur_state = CLEAR;
                }
                break;
            case TRIGGERED:
                if (((flags & ARMF) != 0) && (cur_ms > T)) {
                    cur_state = ALARM;
                } else if ((flags & ARMF) == 0) {
                    cur_state = CLEAR;
                }
                break;
            case ALARM:
```

```
292                     flags |= SIRENF;
293                     if ((flags & ARMF) == 0) {
294                         cur_state = CLEAR;
295                     }
296                     break;
297             }
298 
299             /** update outputs **/
300             sensor1_led();      // Sensor LED 1
301             sensor2_led();      // Sensor LED 2
302             armed_led(cur_ms);  // Armed LED
303             siren(cur_ms);      // Siren
304         }
305     }
306 
```