CS7004 Embedded Systems
Assignment #1 - Ship Alarm

Will Browne
09389822

Problem

1. Design and implement a program to simulate a ship's alarm system on an LPC2468 board making use of a push button that will initiate the alarm signal loop, sounded through the board's speaker device. The alarm consists of 7 short blasts (0.5s) followed by a final long blast (2.5s), with each blast separated by silence (0.5s). The alarm signal is toggled when the push button is held for a period >= 5s.
2. Extend the functionality by allowing a potentiometer to control the duration of the blasts emitted by the speaker.
3. Modify the system to allow for re-entrant exception handlers that makes use of prioritised interrupts.

I will approach each section of the problem in order. I will not mention in this report how each peripheral is configured as that is covered within the comments of the source code.

Design

In order to provide the base functionality I made use of 4 different interrupt sources:

- Push Button (EINT0)
- TIMER0 – 5s counter
- TIMER1 – 0.00383141666667s counter
- TIMER2 – 0.5s counter

Use of timers:

- TIMER0 was used to count for a period of 5s to determine whether or not the alarm signal should begin
- TIMER1 was used in order to provide the actual sound out for the speaker
- TIMER2 was used to control the loop of the alarm signal

Push button and TIMER0

The push button interrupt handler was used in conjunction with TIMER0 to determine whether the button was pressed for a period >= 5s. The push button is initially set to interrupt on rising edges (button down event). When in the interrupt handler for a rising edge, the push button is alternated to interrupt on falling edges and TIMER0 is started. If the button is let go, TIMER0 is stopped and reset to 0.

```
if (!EXTPOLAR) {            /* If button pushed down */
        T0TCR = 0x1;        /* Start TIMER0 */
} else {
        T0TCR = 0x2;        /* Stop and reset TIMER0 */
}
```

```
EXTPOLAR = ~EXTPOLAR;   /* Invert edge mode for EINT0 */
```

If TIMER0 reaches 5s, it will enter its own interrupt handler – triggering the general alarm. Therefore the push button doesn't have to be released in order for the alarm to start.

TIMER0 and TIMER2

Having entered the interrupt handler for TIMER0, we have reached the conditions to start the general alarm. This loop of 7 short blasts followed by a long blast is controlled using TIMER2. TIMER2 is set to interrupt every 0.5s, and in its handler is the functionality to determine whether or not an alarm sound needs to be played, and how long the blast should be.

This is achieved by using some state:

- A '*playing'* Boolean variable, where each time the handler is called the value is inverted (default -> false)
- A '*step'* counter variable to determine where in the alarm sequence we currently are. It increments each time the handler is called (default -> 1 – this is 1 and not 0 because by the time it enters the alarm loop, a single step has already passed)*

I realized that the simplest way to approach this problem was by not controlling how long each note sounds through interrupts, but actually when to turn off the sound output. It is cleaner this way because the silence stays constant (0.5s) throughout the sequence whereas the blast times can vary. Because of this variability, the *step* variable is used to determine when the final blast should be played. This is done through T2MR0 (on the 8th blast/16th step – remember *step* increments every time the handler is called).

Thus every 0.5s, depending on the *playing* variable, sound is output from the speaker (where sound is controlled exclusively by TIMER1 – see next section).

```
if(!playing) {
        T1TCR = 0x1;            /* Start TIMER1 */
} else {
        T1TCR = 0x2;            /* Stop and reset TIMER1 */
}
```

As the lengths of the blasts differ, TIMER2's match register T2MR0 is changed from within the handler.

TIMER1

This is the most simplest of the timers as its sole responsibility is to cause the LPC2468 speaker to output sound. It does this by using a 10-bit sequence of the DACR register. What causes the waveform is the alternation between all 0's and all 1's in this 10-bit sequence, and how frequent that alternation occurs. I chose the note middle C to be the alarm sound, which has a frequency value of 261.6 Hz. In order to configure that to work with the LPC2468 clock, we can divide 261.6 Hz/12,000,000 (since it runs at 12MHz), which results in the figure we need.

TIMER1 and TIMER2

With TIMER2 controlling the alarm loop, TIMER1 is started and stopped depending on the *playing* variable of the interrupt handler.

```
if(!playing) {
        T1TCR = 0x1;            /* Start TIMER1 */
} else {
        T1TCR = 0x2;            /* Stop and rest TIMER1 */
}
playing = !playing;                         // invert state of whether alarm sounds
```

The only remaining functionality for the first part of the assignment was to allow the alarm to be toggled off the same way it was started. This meant adding a condition in the alarm loop handler (TIMER2). Using T2TCR we are able to tell if the alarm loop is active. If the push button is held down for another 5 seconds then all timers must be stopped and reset. As well as this, all state variables must be reverted to their initial values.

```
if(!T2TCR != 0x1) {         /* If alarm loop is not sounding */
        T2TCR = 0x1;        /* Start TIMER2 */
} else {
        T0TCR = 0x2;        /* Stop and reset TIMER0 */
        T1TCR = 0x2;        /* Stop and reset TIMER1 */
        T2TCR = 0x2;        /* Stop and reset TIMER2 */
        init();             /* Reinitialize all variables */
}
```

Adding potentiometer functionality

In the TIMER2 interrupt handler, the value of the potentiometer peripheral (AD0.0) is read through AD0DR0. As this is a 10-bit number (max value of 1023), a arbitrary multiplier value is used to be able to use the potentiometer reading as an offset to TIMER2's duration (IE how long each blast sounds). This means that the potentiometer is read every 0.5s.

Reading potentiometer value and creating offset with reading:

```
potentiometerValue = (AD0DR0 & 0xFFC0) >> 6;
potentiometerValue = potentiometerValue * 5000;
```

Adding offset to match register:

```
T2MR0 = 6000000 + potentiometerValue;
```

Adding re-entrant interrupts functionality

In order to provide functionality for re-entrant interrupts, assembly language was added in an external .s file, whose responsibility was to:

- Clear and acknowledge the interrupt source

- Change to System Mode
- Re-enable IRQs
- Branch to the original interrupt handler in the .c file
- Return to the assembly code to change back to IRQ mode and disable IRQs
- Restore and return from the interrupt.

Each handler vector was changed to point to an external file containing new entry points to each interrupt handler.

Each assembly snippet required some code beforehand:

```
PRESERVE8            ; Ensure 8-bit alignment
AREA Asm, CODE, READONLY
EXPORT *handler_name*
```

The new entry points were made visible in the C source through the *extern* keyword:

```
extern void buttonHandlerARM(void);
extern void alarmSoundHandlerARM(void);
extern void buzzHandlerARM(void);
extern void soundAlarmARM(void);
```

Testing

In order to ensure that my solution worked as outlined in the specification I made various test cases and observed the results

<u>Test cases</u>

Initial functionality:

- Held the push button for a period less than 5s (using stopwatch on my phone to make sure it was accurate) to ensure that nothing occurred.
- Held the push button for a period >= 5s to ensure that the alarm loop began.
- Whilst in the alarm state, holding the push button for a period >= 5s to ensure that the alarm loop stopped.
- Held the push button for a period >=5s after having previously stopped the alarm start to ensure that it would start again and continue as normal from the beginning.
- Held the push button for a period < 5s while the alarm was playing to ensure that the alarm would continue
- Held the push button for a period >= 5s while after having disabled the alarm to ensure it would successfully start again from the beginning
- Held the push button for a period >= 5s while the alarm was playing to ensure that the alarm would disable after having previously disabled and restarted the alarm
- Ensured that even after holding the button for a period of say 3s, letting go, and pressing the button down again would result in still having to hold it for 5s rather than another 2s.

Potentiometer:

- When the alarm was sounding, I turned the potentiometer clockwise to ensure it increases the note play-out duration and decreases when turned anti-clockwise
- When the alarm is sounding observe that the potentiometer doesn't affect the play-out when it its leftmost position
- Started the alarm loop by holding down the push button for >=5s and observing that the note play-out duration didn't vary

Re-entrant Interrupts:
In order to ensure interrupts were re-entrant I had to add some temporary code to my alarm loop handler which is triggered following a button down for >= 5s. The idea was to add an infinite loop that would be triggered when the condition for the alarm to turn off was true (button held for >=5s during alarm loop). As well as this the priority was changed for each handler:

Sound interrupt **>** Alarm Loop interrupt **>** Button Down interrupt

This meant that even if holding down the push button for >=5s during an alarm loop, it would continue because the priority of the loop would allow it to interrupt the execution of the infinite loop.

Conclusion
I have tested my implementation as well as I could, and I'm happy that it functions according to the specification.