Assignment 3 Report

ECEN-449-500
Microprocessor System Design

Assignment 3
Lab Report: Interrupt System

Name: Selman Tabet
UIN: 724009589

Electrical and Computer Engineering
Texas AM University at Qatar
Spring 2021

Overview

This assignment required the usage of an EA LPC4088 QuickStart Board mounted on a base board through an FFC cable (basically a ribbon cable, but smaller). Before the board was connected to the computer through a USB 2.0 Micro cable. The online Mbed IDE was used to compile the C code and the output binary files were then copied into the device's flash storage through USB, and leveraging the built-in HDK USB drag-n-drop programming feature to automatically flash the binary file into the board's memory.

This assignment focuses on the usage of the board's interrupt system by using the on-board pushbutton as a trigger to call an Interrupt Service Routine (ISR); a program that is executed upon a defined signal, through hardware or software. Often being unforeseeable in nature, ISRs are different from subroutines in the way they start. The developer would be aware of exactly when and at which step of the program a subroutine would be called, while an ISR is programmed separately such that it runs when an external and unpredictable, yet defined trigger is signaled. In this assignment, a button press and a button release - which are hardware signals - are the triggers that would be used to call ISRs throughout the programs included here.

Question 1

```
/* Selman Tabet (@selmantabet - https://selman.io/) - UIN 724009859
Assignment 3 - Question 1

This function turns the LEDs in the sequence 4-1-2-3. When the user presses
the pushbutton, all LEDs would simultaneously turn on for three seconds before
resuming the previous sequence.

ISR calls here cannot stack up as nested ISRs do, meaning that when an ISR
is running, all further interrupt signals are ignored until ISR exit.
In this case, that means that the pressing the button multiple times in
succession would not extend the "All LEDs ON" state.

Special functions that protect the integrity of the LED states prior to
main process interruption were designed.

Developed using the Mbed IDE. Tested on an EA LPC4088 QuickStart Board. */

#include "mbed.h"

//Create DigitalOut objects to control LED1-LED4:
DigitalOut my_led1(LED1); //Active Low
DigitalOut my_led2(LED2); //Active Low
DigitalOut my_led3(LED3); //Active High
DigitalOut my_led4(LED4); //Active High

//Create an InterruptIn object for the pushbutton:
InterruptIn Button(p23);

/*The following two functions work on saving the LED states prior to
jumping into an ISR, then loaded back before RETI (Return from Interrupt),
this was implemented so that the LED states remain intact following each ISR.
The ISRs modify the LED states, so it is imperative to maintain
their integrity in the principal routine for a more graceful execution.*/

int encode_state(DigitalOut led1, DigitalOut led2,
                 DigitalOut led3, DigitalOut led4)
                 {
                 /*The function saves the current LED states for
                 future restoration. The function is operating-mode-agnostic
                 i.e. it does not matter if the LEDs operated in
                 Active-Low or Active-High modes*/
                     int state_seq = 0b0000; //Bit sequence led1,led2,led3,led4
                     if (led1 != 0) state_seq |= 0b1000;
```

```
                    if (led2 != 0) state_seq |= 0b0100;
                    if (led3 != 0) state_seq |= 0b0010;
                    if (led4 != 0) state_seq |= 0b0001;
                    return state_seq;
                }

void decode_state(int state_id){
    /*This function decodes and restores the LED states back to their
    original form.
    Assumed that the LEDs are declared externally. !!Non-modular function!! */
    my_led1 = (state_id & 0b1000) >> 3;
    my_led2 = (state_id & 0b0100) >> 2;
    my_led3 = (state_id & 0b0010) >> 1;
    my_led4 = (state_id & 0b0001);
    }

void pushbutton_isr(void){
        int save_state = encode_state(my_led1, my_led2, my_led3, my_led4);
        my_led1 = 0; my_led2 = 0; my_led3 = 1; my_led4 = 1; wait(3); //All ON.
        decode_state(save_state); //Load pre-ISR state

}

int main(){
    my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; //All OFF.
    Button.mode(PullUp); //Setup a PullUp Resister
    Button.fall(&pushbutton_isr); //Register an ISR on the fall edge

    while(1) { //LED4-LED1-LED2-LED3 sequence.
        my_led4 = 1; wait(0.75); my_led4 = 0;
        my_led1 = 0; wait(0.75); my_led1 = 1;
        my_led2 = 0; wait(0.75); my_led2 = 1;
        my_led3 = 1; wait(0.75); my_led3 = 0;
    }
}
```

This function turns on and off one LED at a time in the order `LED4 -> LED1 -> LED2 -> LED3` with an interval of 750ms in between each LED. The button was setup to interrupt execution and call an ISR that would turn all the lights on simultaneously for three seconds before returning to the principal routine. The pushbutton interrupt is triggered on signal fall as the button's value is zero when pressed, therefore it falls from a `HIGH` to a `LOW` value when being pressed, and vice versa for signal rise trigger in the case of a button release. No interrupt signal was defined here for the case of a signal rise, but it will be used in another program.

The ISRs in this case make changes to the LED states, and since interrupts can be done through any stage of the program, the original LED states would be lost once changes are made in a different routine (an ISR). Indeed, the program counter containing the next instruction would be pushed into the stack and retrieved to continue execution, but that does not take into account external value changes (similar to data marked with the `volatile` qualifier in C) that could potentially happen during an ISR, it only preserves execution stage. A bug attributed to this was observed when testing the interrupt system where one of the LEDs was stuck for an interval awaiting to be turned off in a distant upcoming instruction.

To protect the integrity of the LED states (AKA "original context") prior to ISR execution, a pair of special functions that addresses the issue were designed and implemented. In a nutshell, a function saves the last state of the LEDs into state bits, and once the ISR is over and about to exit, the state bits are used in another function to load the LED values back into their respective digital outputs, effectively restoring the LED states back to their pre-ISR form and continuing the principal routine as if there was never an interruption. The two functions are aptly called `encode_state` and `decode_state`, feel free to refer to them as `save_state` and `load_state` or what have you. The functions are operating-mode-agnostic (OMA for short, maybe?), meaning that they do not take into account the LEDs' operating modes, since the state bits are copies from the original LED values, and are dealt with on an as-is basis.

It is also important to briefly talk about successive interrupts. ISR calls here cannot stack up as nested ISRs do, meaning that when an ISR is running, all further interrupt signals are ignored until ISR exit. The quality and tactility of the pushbutton is also important in the context of interrupt signals as a flimsy-feeling and improperly designed switch could result in multiple interrupt signals being generated involuntarily. There are many technologies surrounding press sense methods implemented by different companies, from Alps to Cherry MX, all addressing concerns with accidental button presses by enhancing the tactility and feel of the buttons for the end-user.

Many of the following programs employ the same algorithms described here, and thus there is no need to re-discuss the approaches to context saving and nested ISRs later on.

Question 2

```
/* Selman Tabet (@selmantabet - https://selman.io/) - UIN 724009859
Assignment 3 - Question 2

This function turns the LEDs in the sequence 1-2-4-3. When the user presses
the pushbutton, all LEDs would simultaneously turn on for a second, then off
for two seconds before resuming the main sequence.
When the button is released *during the main sequence*, LED4 and LED2 would
alternatively turn on for 5 seconds before resuming the main sequence.

ISR calls here cannot stack up as nested ISRs do, meaning that when an ISR
is running, all further interrupt signals are ignored until ISR exit.

Special functions that protect the integrity of the LED states prior to
main process interruption were designed.

Developed using the Mbed IDE. Tested on an EA LPC4088 QuickStart Board. */

#include "mbed.h"

//Create DigitalOut objects to control LED1-LED4:
DigitalOut my_led1(LED1); //Active Low
DigitalOut my_led2(LED2); //Active Low
DigitalOut my_led3(LED3); //Active High
DigitalOut my_led4(LED4); //Active High

//Create an InterruptIn object for the pushbutton:
InterruptIn Button(p23);

/*The following two functions work on saving the LED states prior to
jumping into an ISR, then loaded back before RETI (Return from Interrupt),
this was implemented so that the LED states remain intact following each ISR.
The ISRs modify the LED states, so it is imperative to maintain
their integrity in the principal routine for a more graceful execution.*/

int encode_state(DigitalOut led1, DigitalOut led2,
                 DigitalOut led3, DigitalOut led4)
                 {
                 /*The function saves the current LED states for
                 future restoration. The function is operating-mode-agnostic
                 i.e. it does not matter if the LEDs operated in
                 Active-Low or Active-High modes*/
                     int state_seq = 0b0000; //Bit sequence led1,led2,led3,led4
                     if (led1 != 0) state_seq |= 0b1000;
```

```
                    if (led2 != 0) state_seq |= 0b0100;
                    if (led3 != 0) state_seq |= 0b0010;
                    if (led4 != 0) state_seq |= 0b0001;
                    return state_seq;
                }

void decode_state(int state_id){
    /*This function decodes and restores the LED states back to their
    original form.
    Assumed that the LEDs are declared externally. !!Non-modular function!! */
    my_led1 = (state_id & 0b1000) >> 3;
    my_led2 = (state_id & 0b0100) >> 2;
    my_led3 = (state_id & 0b0010) >> 1;
    my_led4 = (state_id & 0b0001);
    }

void pushbutton_isr_press(void){ //Button Press ISR
        int save_state = encode_state(my_led1, my_led2, my_led3, my_led4);
        my_led1 = 0; my_led2 = 0; my_led3 = 1; my_led4 = 1; //All ON.
        wait(1);
        my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; //All OFF.
        wait(2);
        decode_state(save_state); //Load pre-ISR state
}

void pushbutton_isr_release(void){ //Button Release ISR
        int save_state = encode_state(my_led1, my_led2, my_led3, my_led4);
        my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; //Clear LEDs.
        /*Without using time(), we leverage the fact that the behaviour
        is well-defined and definite.*/
        double counter = 0.0;
        while (counter < 5.0){ //5 is not divisible by 0.75 -> will exceed limit
            my_led4 = 1; wait(0.75); counter += 0.75; my_led4 = 0;
            my_led2 = 0; wait(0.75); counter += 0.75; my_led2 = 1;
        }
        decode_state(save_state); //Load pre-ISR state

}


int main(){
    my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; //All OFF.
    Button.mode(PullUp); //Setup a PullUp Resister
    Button.fall(&pushbutton_isr_press); //Falling edge = Press ISR
```

```
    Button.rise(&pushbutton_isr_release); //Rising edge = Release ISR

    while(1) { //LED1-LED2-LED4-LED3 sequence.
        my_led1 = 0; wait(0.5); my_led1 = 1;
        my_led2 = 0; wait(0.5); my_led2 = 1;
        my_led4 = 1; wait(0.5); my_led4 = 0;
        my_led3 = 1; wait(0.5); my_led3 = 0;
    }
}
```

Here the board's LEDs turn on and off in the `LED1 -> LED2 -> LED4 -> LED3` sequence, with an interval of 500ms between each LED. A button push (signal fall) would cause all the LEDs to turn on simultaneously for a second, then turn off for two seconds before resuming the principal routine. A button release (signal rise) results in `LED1` and `LED3` turning off, while `LED4` and `LED2` flash in an alternating manner with 750ms in between switches. This alternating behavior lasts for five seconds before returning to the principal routine.

The implementation did not use the `time.h` library for the sake of simplifying the implementation, the one issue to note is that the five second threshold will most definitely be exceeded for two reasons. The first reason is because of the usage of an accumulator that increments on each `wait()` command, which adds a few instructions worth of overhead. Most significantly, however, is the fact that 5 is not divisible by 0.75, which means that the LED can alternate either six times to hit the 4.5s mark, or exceed the threshold by switching for a seventh time to hit the 5.25s mark then break the `while` loop. The latter scenario was selected for the implementation.

Question 3

```
/* Selman Tabet (@selmantabet - https://selman.io/) - UIN 724009859
Assignment 3 - Question 3

This function turns the LEDs in the sequence 1-2-4-3. When the user presses
the pushbutton, the LED sequence would run at a faster rate until the user
releases the button. Upon button release, all LEDs would turn off for
three seconds prior to resuming the main sequence.

ISR calls here cannot stack up as nested ISRs do, meaning that when an ISR
is running, all further interrupt signals are ignored until ISR exit.
In this case, pressing the button again while the LEDs are off from the last
registered button release would not cause anything.

Special functions that protect the integrity of the LED states prior to
main process interruption were designed.

Developed using the Mbed IDE. Tested on an EA LPC4088 QuickStart Board. */

#include "mbed.h"
double interval = 1.0 ; //Time between LED state switches
bool  flag = false;

//Create DigitalOut objects to control LED1-LED4:
DigitalOut my_led1(LED1); //Active Low
DigitalOut my_led2(LED2); //Active Low
DigitalOut my_led3(LED3); //Active High
DigitalOut my_led4(LED4); //Active High

//Create an InterruptIn object for the pushbutton:
InterruptIn Button(p23);

/*The following two functions work on saving the LED states prior to
jumping into an ISR, then loaded back before RETI (Return from Interrupt),
this was implemented so that the LED states remain intact following each ISR.
The ISRs modify the LED states, so it is imperative to maintain
their integrity in the principal routine for a more graceful execution.*/

int encode_state(DigitalOut led1, DigitalOut led2,
                 DigitalOut led3, DigitalOut led4)
                 {
                 /*The function saves the current LED states for
                 future restoration. The function is operating-mode-agnostic
                 i.e. it does not matter if the LEDs operated in
```

```
                Active-Low or Active-High modes*/
                    int state_seq = 0b0000; //Bit sequence led1,led2,led3,led4
                    if (led1 != 0) state_seq |= 0b1000;
                    if (led2 != 0) state_seq |= 0b0100;
                    if (led3 != 0) state_seq |= 0b0010;
                    if (led4 != 0) state_seq |= 0b0001;
                    return state_seq;
                }

void decode_state(int state_id){
    /*This function decodes and restores the LED states back to their
    original form.
    Assumed that the LEDs are declared externally. !!Non-modular function!! */
    my_led1 = (state_id & 0b1000) >> 3;
    my_led2 = (state_id & 0b0100) >> 2;
    my_led3 = (state_id & 0b0010) >> 1;
    my_led4 = (state_id & 0b0001);
    }

void pushbutton_isr(void){
    int save_state = encode_state(my_led1, my_led2, my_led3, my_led4);
    if (flag == true){
        my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; wait(3); //All OFF.
        interval = 1.0;
    }
    else{
        interval = 0.25;
    }
    flag = !flag; //Invert flag
    decode_state(save_state); //Load pre-ISR state
}


int main(){
    my_led1 = 1; my_led2 = 1; my_led3 = 0; my_led4 = 0; //All OFF.
    Button.mode(PullUp); //Setup a PullUp Resister
    Button.fall(&pushbutton_isr); //Falling edge = Button Press -> ISR
    Button.rise(&pushbutton_isr); //Rising edge = Button Release -> ISR

    while(1) { //LED1-LED2-LED4-LED3 sequence.
        my_led1 = 0; wait(interval); my_led1 = 1;
        my_led2 = 0; wait(interval); my_led2 = 1;
        my_led4 = 1; wait(interval); my_led4 = 0;
        my_led3 = 1; wait(interval); my_led3 = 0;
```

```
    }
}
```

This function turns the on-board LEDs on then off, with the exact same sequence as last time; `LED1 -> LED2 -> LED4 -> LED3`. The interval between each LED state, however, is one second. Upon a button press (signal fall), the LEDs would start the same sequence, but with an interval of 250ms between state transitions. When the user releases the button (signal rise), all the LEDs will be turned off for three seconds prior to resuming the principal routine.

The ISR simply switches two global variables; a boolean `flag` and a `double` variable corresponding to the interval value that controls the speed at which the LED sequence runs according to the user's action. Both interrupt triggers (rise and fall) call the same ISR, though in a non-preemptive manner. On the first trigger - a button press - the speed ramps up (or the interval shortens) and the `flag` is set, once the second trigger is detected - a button release - the `flag` is found to be tripped and the LEDs turn off for three seconds before restoring the original speed and clearing the `flag`. Because of the non-preemptive nature of the ISR, the second trigger's routine, which lasts just over three seconds, cannot be interrupted by a subsequent signal fall trigger. The fact that this also ensures that the flag is cleared prior to executing another ISR is the icing on the cake, as it reassures that the intended behavior is maintained and not flipped upside down.