UCD School of Electrical and Electronic Engineering

EEEN40280

Digital and Embedded Systems

Microcontroller & Assembly Language Report

Name: Cuan De Búrca **Student Number:** 17398541

Name: Dylan Boland **Student Number:** 17734349

Date: 03 February 2022 **Brightspace Group Number:** 5

By submitting this report through Brightspace, we certify that ALL of the following are true:

1. We have read the *UCD Plagiarism Policy* (available on Brightspace). We understand the definition of plagiarism and the consequences of plagiarism.
2. We recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the documents above.
3. We have not plagiarised any part of this report. The work described was done by the team, and this report is all our own original work, except where otherwise acknowledged in the report.

Introduction

In this assignment, we were asked to generate a square wave output signal on a pin of the ADuC841 microcontroller, first using a software delay, then using a hardware timer and interrupts.

Task 1: Software Timing

The algorithm is based on the blinking LED program provided as an example. The program changes the state of the output pin, waits for a suitable delay time, and repeats. The delay subroutine for this task only requires two loops instead of three which was used for the blinking LED example program as the frequency is much higher which calls for a shorter delay.

The frequency required for our group is 1200 Hz. We calculated that to be 9216 clock cycles at our clock frequency of 11.0592 MHz. So we wanted to change the state of the output pin every 4608 clock cycles. To achieve that in a software delay, we needed two 8-bit counters.

Program Design

The main program has only 3 instructions, following the algorithm above exactly. It calls a delay subroutine to get the required time delay. A snippet of code showing this subroutine is shown below:

```
;-------------------------------------------------------------------
        ; SUBROUTINES
DELAY:  MOV     R6, #8      ; outer loop repeats 8 times  (2 clk cycles)
DLY1:   MOV     R7, #189    ; load in 189
        DJNZ    R7, $       ; inner loop, delay of (189*3 + 2 + 5 = 574 clk cycles)
        NOP
        NOP
        DJNZ    R6, DLY1    ; outer loop, (8*574 + 2 = 4594 clk cycles)
        NOP
        NOP
        RET                 ; return from subroutine (4 clk cycles)

;-------------------------------------------------------------------
```

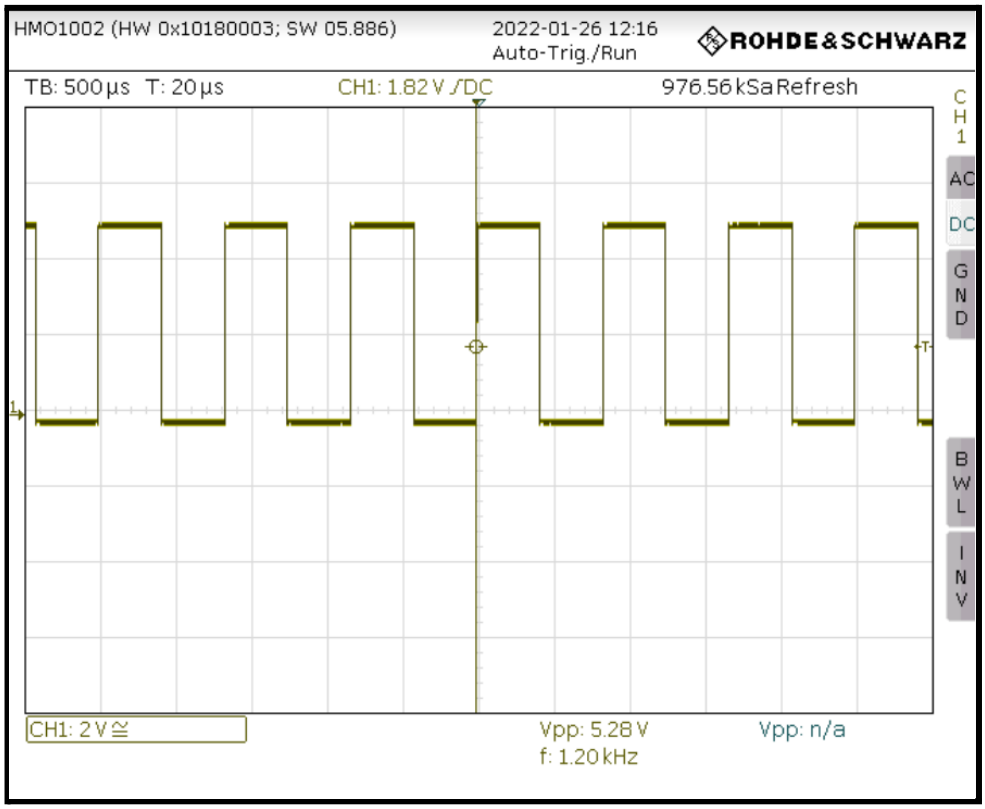
In the delay subroutine, we used register R7 as the inner counter, starting at a value of 189. This was decremented in a loop using only the DJNZ instruction, which uses 3 clock cycles. So it reaches 0 after 567 clock cycles. Once this happens, two NOP instructions are executed taking 2 clock cycles before the value in the outer register R6 is checked (taking 3 clock cycles) and it is not zero, we jump back to the DLY1 label which loads R7 with 189 again (taking 2 clock cycles). All in all, by the time the inner loop completes and resets it takes a total of 574 clock cycles.

The outer loop runs 8 times, so the complete DELAY subroutine takes 4600 clock cycles, which includes the 2 clock cycles used in loading in the R6 register value, and the 4 clock cycles used in executing the RET instruction. Adding in the 8 clock cycles used in the MAIN program, we obtain the 4608 clock cycles we need to delay the complementing of the pin at P3.6 to obtain our desired frequency of 1200 Hz.

With that number of clock cycles in each loop, we expected an output frequency of 1200 Hz.

Result:

We measured a frequency of 1200.086 Hz using the frequency meter. This is an error of 0.007% from what we expected, which was the target value of 1200 Hz. We deemed this to be acceptable. We used both the oscilloscope in the lab and the frequency meter to measure the frequency which was actually generated in the lab. A screenshot from the oscilloscope showing the square wave and measured frequency is included below:



(1200 Hz signal from oscilloscope)

Example Program

We calculated an output frequency to be 674.56 Hz using the calculations below:

$$0.5T = \frac{2^{13}}{11.052 \times 10^6}$$
$$\therefore T = 0.001482446$$
$$\therefore f = \frac{1}{T} = 674.56Hz$$

The reason that 2^13 is in the numerator is because a 13-bit counter is used to generate the overflow (Timer is set to Mode 0). When we measured this frequency in the lab using the frequency reader, we found it to be 675.048 Hz.

Task 2: Hardware Timing

We chose to use an algorithm based on suggestion A in the instructions, as the eight frequencies which we chose to generate were not related to each other in a simple way i.e. not multiples of the others, etc. due to the fact that our target frequencies were musical notes

spaced with intervals of thirds: A (880 Hz), C (1046.50 Hz), E (1318.51 Hz), G (1567.98 Hz), B (1975.53 Hz), D (2349.32 Hz), F (2793.83 Hz), A (3520 Hz).

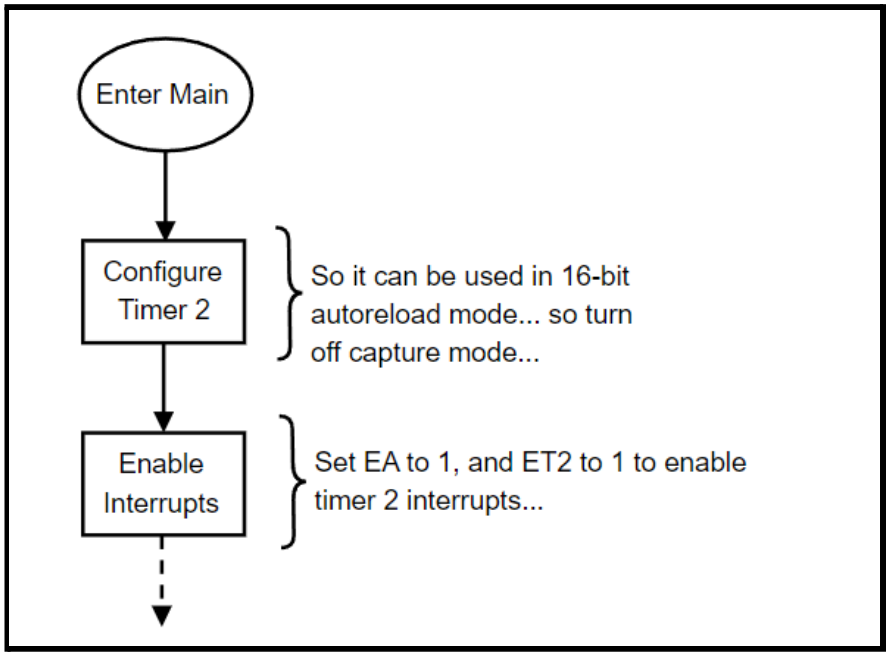
Because of this we decided the most appropriate way to generate these (or very close to these) frequencies would be to load a reload value into Timer 2 which would cause an overflow every half period of our chosen desired frequency, and then complement the value of P3.6 which is the in on which our square wave is outputted. We picked Timer 2 to generate the interrupts, because it has a facility to set a 16-bit reload value which is very convenient for loading in appropriate values to the timer for our method.

We used switches connected to P2.0, P2.1 and P2.2, which give 8 combinations to select the frequency. The frequencies are shown in Table 1 below, along with the corresponding values from the switches.

Program Design

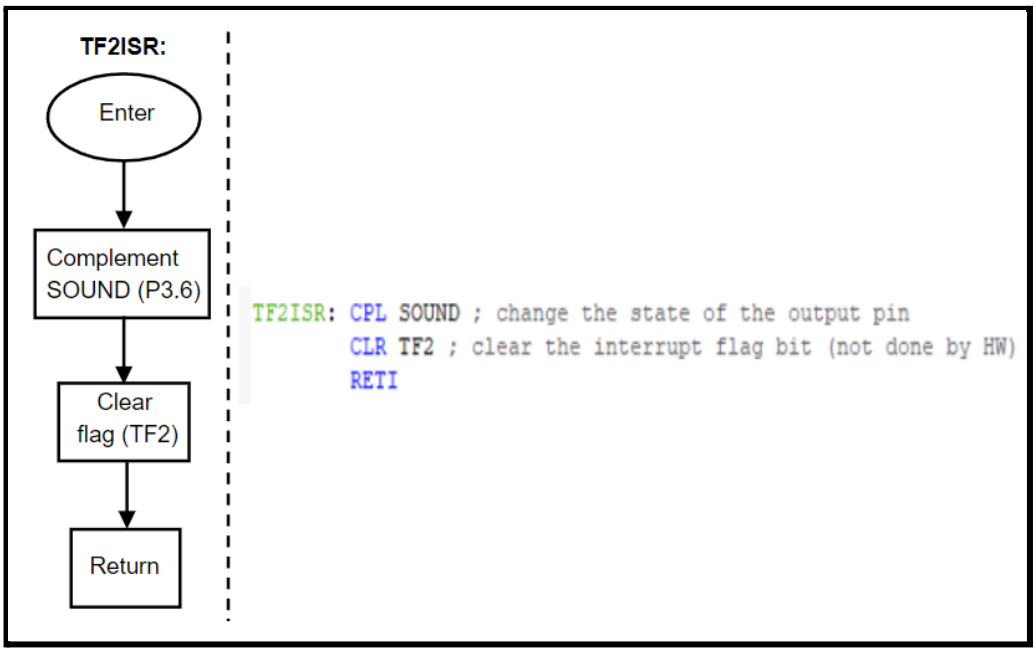
Main:

In the main part of our program, timer 2 is configured and its interrupts are enabled. This is done by loading values into the T2CON and IE registers:



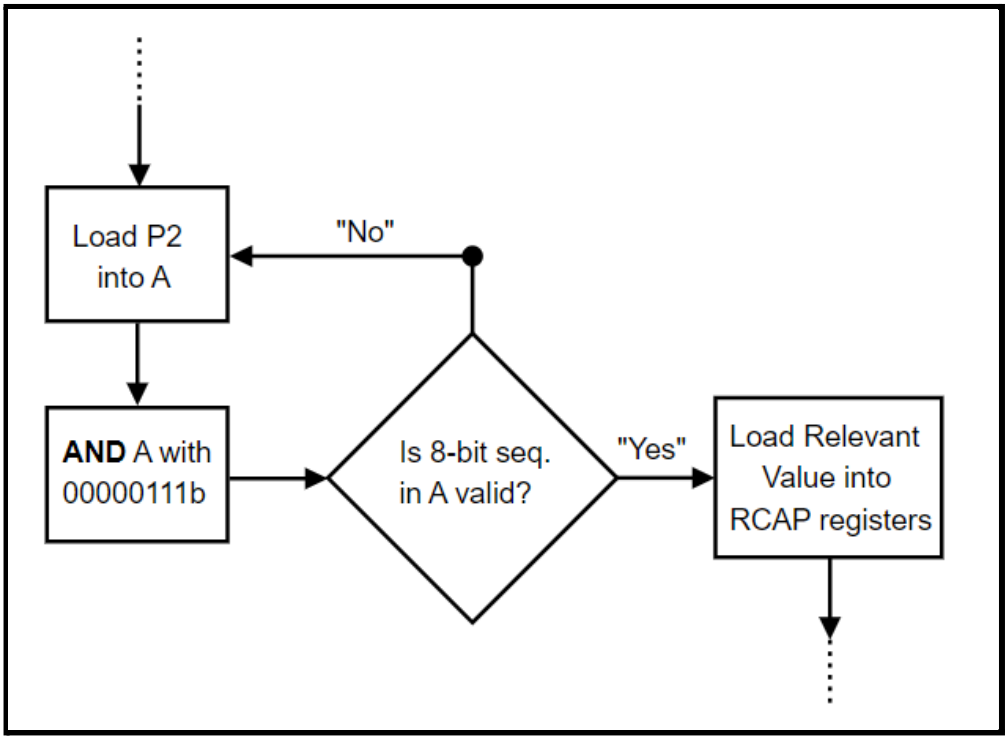
The Interrupt Service Routine (ISR) for Timer 2:

The interrupts service routine for timer 2 is short. First, it complements P3.6, which is the bit corresponding to SOUND. Then, before returning, it clears the interrupt flag as this is not done automatically by the hardware.



Reading the values from the Switches on Port 2:

In order to read the state of the three relevant switches on port 2, all eight bits of port 2 are loaded into the accumulator (A). Once there, a logical bit-wise AND operation is performed on these eight bits with 00000111. This has the effect of suppressing the five most significant bits of the byte in the accumulator. These five bits correspond to the switches which are not being used to control the frequency generated. The flowchart below hopefully depicts this idea clearly:



As is hopefully clear from the above, the 8-bit sequence is then used to decide what values to load into RCAP2L and RCAP2H (referred to as the RCAP registers in the above flowchart).

If the value in the accumulator is changed unexpectedly (e.g. by the triggering of an interrupt of a different program which uses the accumulator) so that its contents do not match with any of the eight possible values, then we loop back to the top block, where the port 2 values are loaded into the accumulator.

Shown below is a snippet of the code:

Code snippet:

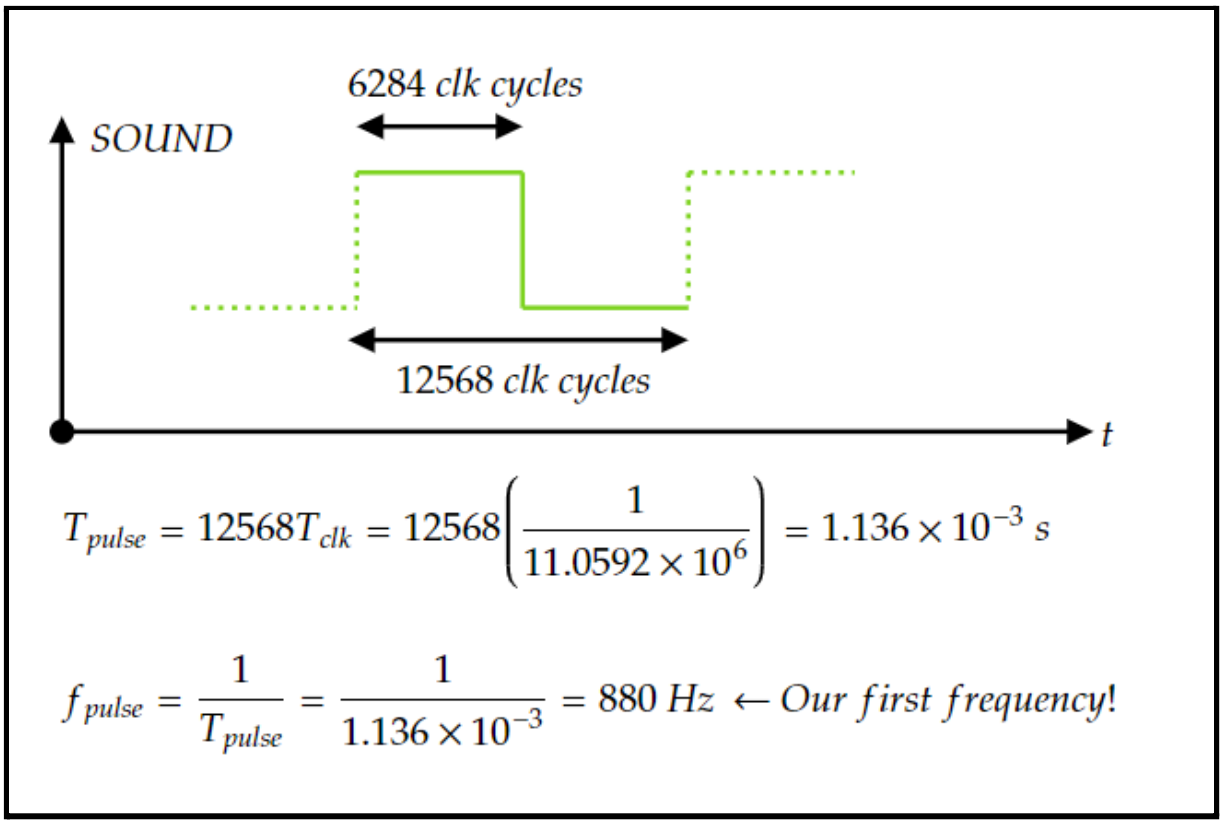
```
LOAD:   MOV     A, P2
        ANL     A, #00000111b
FREQ1:  CJNE    A, #00000000b, FREQ2
        MOV     RCAP2H, #0E7h
        MOV     RCAP2L, #74h
        MOV     P0, #11111110b
        RET

FREQ2:  CJNE    A, #00000001b, FREQ3
        MOV     RCAP2H, #0EBh
        MOV     RCAP2L, #5Ch
        MOV     P0, #11111101b
        RET
```

...loading in E774H, or 59252 in decimal

Turning on the right LED in each case...

For example in the FREQ1 subroutine we load in 231 (E7 in Hex code) into RCAP2H, and we load in 116 (74 in Hex code) into RCAP2L. This 16-bit value (59252) is loaded into the 16-bit counter of timer 2. Timer 2 then counts from 59252 to 65535 (i.e., 2^{16} - 1), and overflows after a total of 6284 clock cycles. 6284 clock cycles corresponds to a frequency of 880 Hz, which was our first frequency:



$$T_{pulse} = 12568 T_{clk} = 12568\left(\frac{1}{11.0592 \times 10^{6}}\right) = 1.136 \times 10^{-3}\ s$$

$$f_{pulse} = \frac{1}{T_{pulse}} = \frac{1}{1.136 \times 10^{-3}} = 880\ Hz \leftarrow Our\ first\ frequency!$$

To generate the other frequencies we simply load in different values into RCAP2H and RCAP2L in order to control the rate at which the counter overflows. As is also mentioned in the code snippet above, for each of the eight frequencies a different 8-bit one-hot code is loaded into port 0 in order to turn on an LED. The leftmost LED was used for the lowest of our frequencies (880 Hz), and each subsequent LED to the right was used for a higher and higher frequency, with the rightmost LED being on for our maximum frequency of 3520 Hz.

Improvement:

In order to implement the extra feature, the following things were done:

- P3.2 was configured to be an input pin.
- External interrupt 0 was enabled by setting the IE0 bit in the TCON register.
- The trigger type was set to be edge-sensitive so that the interrupt service routine (IE0ISR) only triggered on a high-to-low transition of P3.2.
- The F0 flag bit in the Program Status Word register was used to control when the LED was off, or flashing as normal.

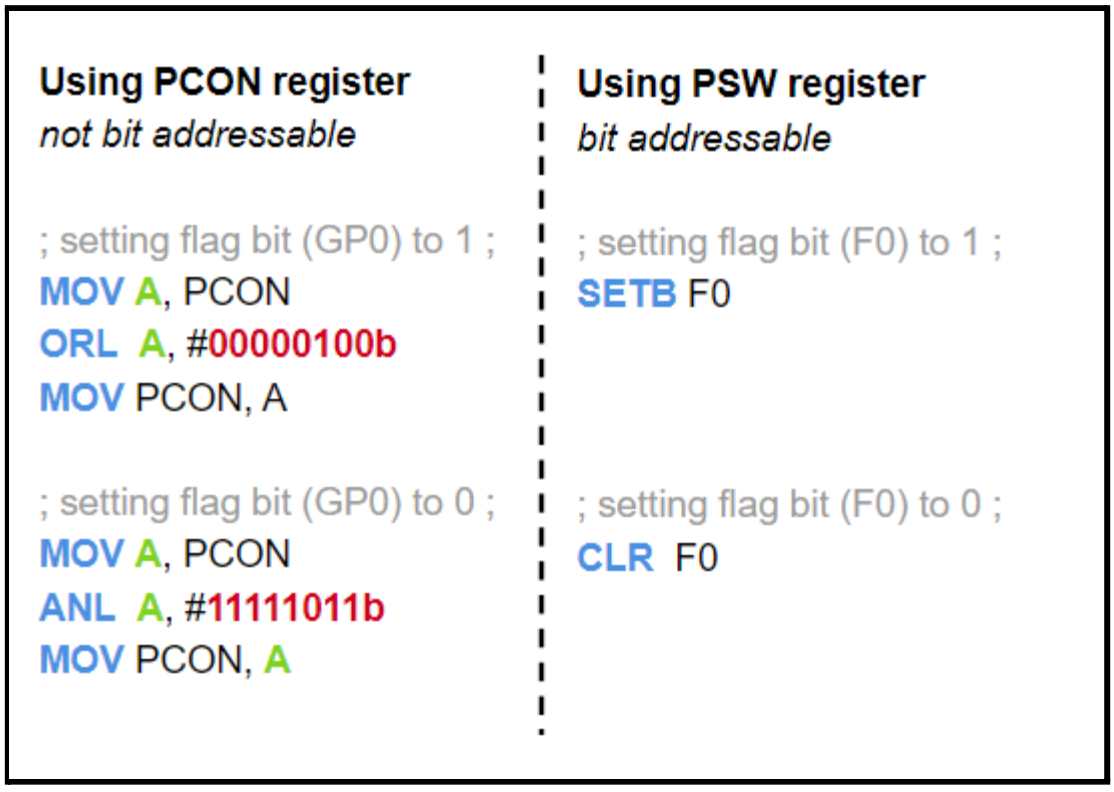
Why use a single flag bit?

Since the LED was either going to be flashing or not flashing, a single flag bit seemed sufficient - using a full 8-bit register to store a “0” or a “1” appeared wasteful.

Why use the Program Status Word (PSW) register?

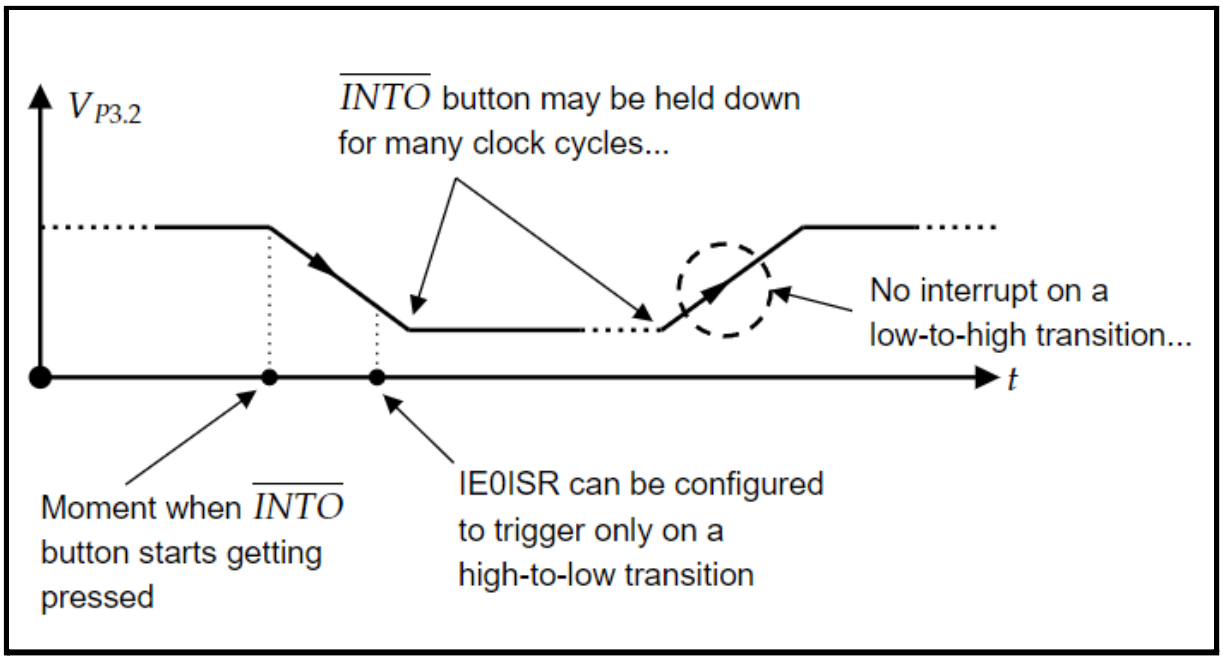
Initially, we thought of using the PCON register, as it has two general-purpose flag bits: GF0 and GF1. We switched to the PSW register for two main reasons:

- The PCON register is not bit addressable, whereas the PSW register is: this means we can operate on the bits directly.
- Performing operations on the byte in the PCON register would take more clock cycles and so be more inefficient: we would need to load the byte into the accumulator and make sure not to change the other important bits (which might be used by other pieces of hardware or software) by performing ORL and ANL operations carefully - hopefully the diagram below makes this more clear:



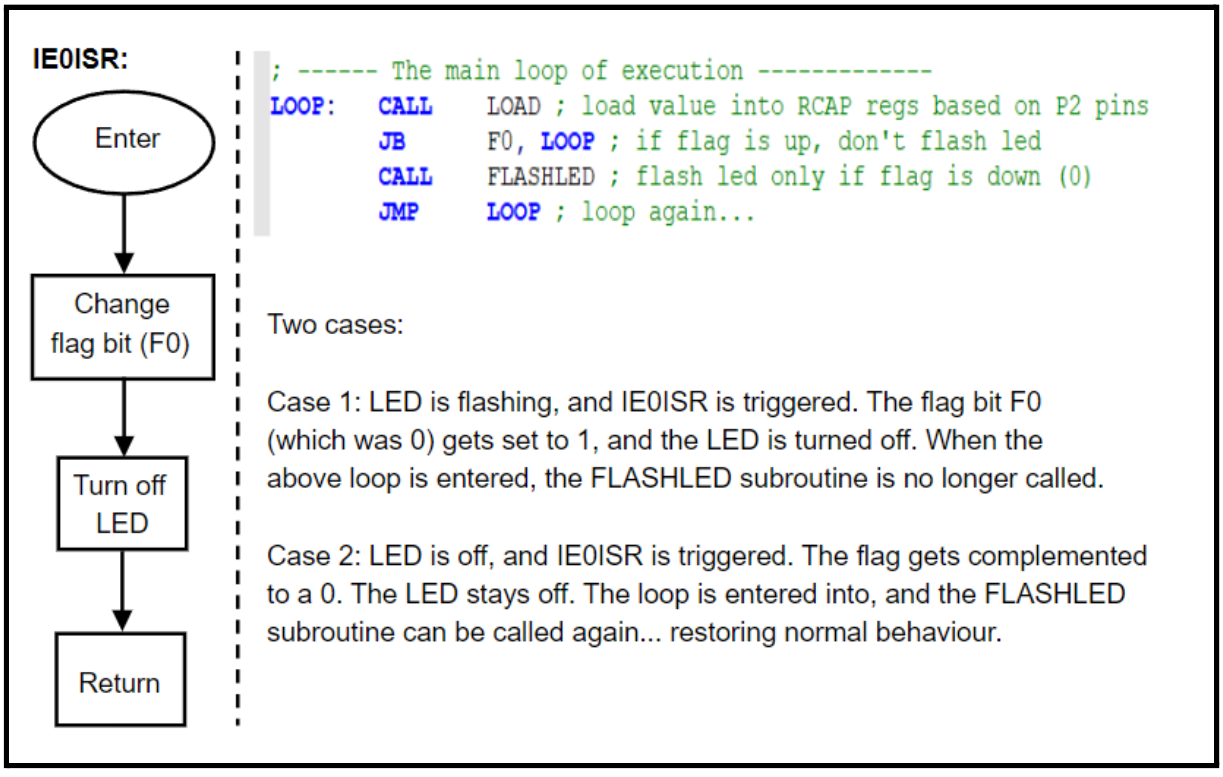
Why set the trigger type to be edge-sensitive?

Since the duration of time for which the $\overline{INT0}$ button is pressed will be long relative to a clock cycle, we did not want to use level-sensitive detection. We instead chose to use edge-sensitive detection, so the external interrupt was only triggered when $\overline{INT0}$ made a high-to-low transition. Hopefully the diagram below is clear:



The Algorithm and the Interrupt Service Routine (ISR):

While planning how the ISR would function, we kept in mind that it should be short and do only what is required. The diagram below will hopefully demonstrate what happens when the interrupt is handled:

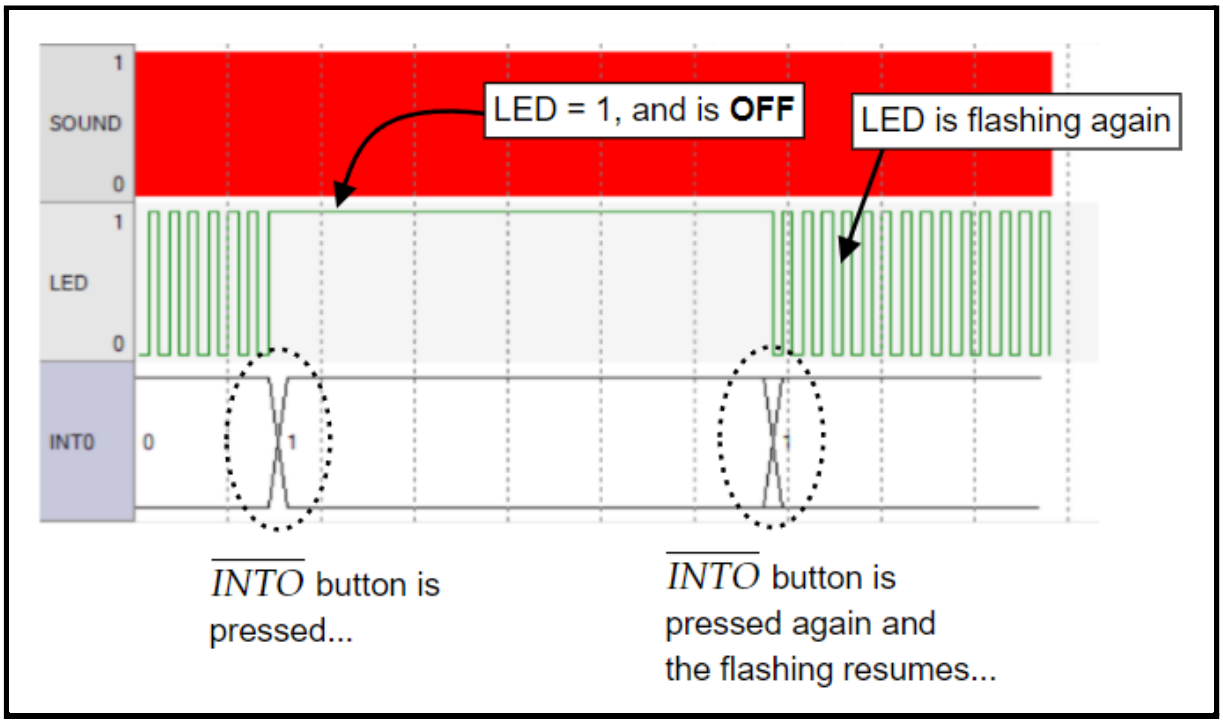


An important point is that during the setup stage, the flag bit F0 is set to 0 using the CLR instruction.

Below is a snippet of code, showing the ISR for IE0:

```
137   IE0ISR: CPL F0
138           SETB LED
139           RETI
```

The figure below shows the results from the simulation performed in the Keil µVision debugger:



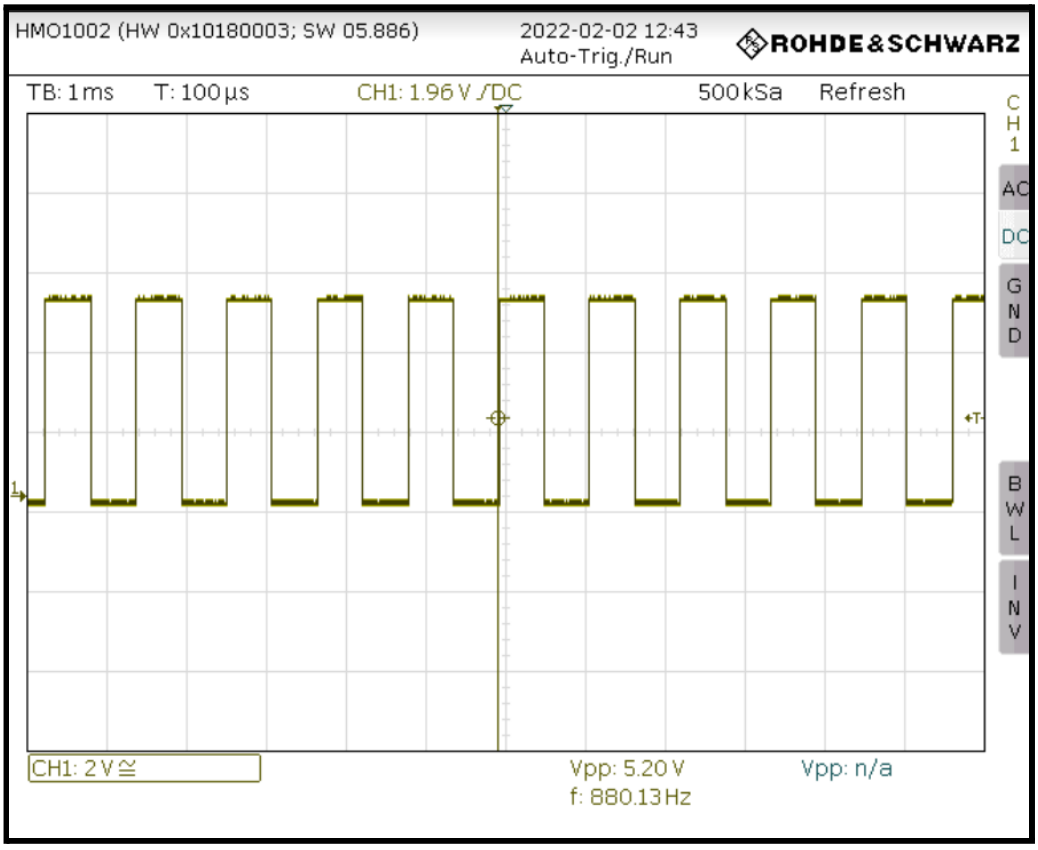
As can be seen above, the sound signal does not get interrupted (its frequency is much higher than that of the flashing LED, and so its waveform looks like a solid red block in the above figure). Unfortunately we did not have time to test this extra feature on the hardware, where the phenomenon of bounce would be a problem. Consulting online resources, it seems the problem of switch bounce may be combated using hardware or software.

One basic idea might be to wait many clock cycles after a transition is detected before acting. When the button is intentionally pressed, it should stay at the same level for many, many clock cycles. When it is bouncing, its level will be changing more rapidly.

Results

We implemented the LEDs on port 0 which use one-hot code to display which frequency is being generated - we were happy with this as it changes at the same time as the frequencies changed. We also modified the code in the blink.asm example file to drive our ‘heartbeat LED’ on P3.6. We were pleased with the result as it flashed at a rate that was visible to the eye when the program was running on the microcontroller.

We also obtained screenshots of each of the frequencies generated on the oscilloscope. As an example, a screenshot from the oscilloscope of the 880 Hz signal is shown below:



(880 Hz signal from oscilloscope)

Results Table:

Switch Value	Target Frequency (Hz)	Period (clock cycles)	Reload Value	Expected Frequency (Hz)	Measured Frequency (Hz)
0	880	12568	59252	879.9	880.0
1	1046.5	10568	60252	1046.4	1046.5
2	1318.51	8388	61342	1318.4	1318.5
3	1567.98	7054	62009	1567.7	1567.9
4	1975.53	5598	62737	1975.5	1975.7
5	2349.32	4708	63182	2349.0	2349.1
6	2793.83	3958	63557	2794.1	2794.3
7	3520	3144	63965	3519.7	3520.0

Conclusion

In conclusion, we were able to meet the core specifications. The work we undertook was to:

- Produce a 1200 Hz pulse using a software delay.
- Using a hardware timer and interrupts, produce eight different frequencies. The correct LED was also turned on for each frequency.
- Create a flashing LED which acts as a heartbeat indication.

We were also able to modify the code so as to facilitate the extra feature involving the $\overline{INTO}$ pushbutton. This involved creating another interrupt service routine (IE0ISR), and making use of a flag bit (F0) in the PSW register. This extra feature also entailed the changing of our main loop. The simulation of the correct functioning of the $\overline{INTO}$ pushbutton in the Keil debugger is promising, although we will need to briefly test it on the hardware at the beginning of the next laboratory session. (A graph of this simulation can be found near the “Improvement” section above)