ACE411 - Embedded Systems

LAB 1 - A Simple Timer

Odysseas Stavrou 2018030199
October 2021
Technical University of Crete
Prof.: A. Dollas

In the first Lab Excercise of the course on Embedded Systems (ACE411), the goal was to create a 20ms Timer Using two methods, Interrupts and Polling (loops). The language of Implementation was AVR (Alf and Vegard's RISC / Advanced Virtual RISC) Assembly.

1.) Timer Using Two Nested Loops and Polling

For creating the timer using loops, two nested loops were required because of the 8-bit resolution of the microprocessor. The inner loop repeats 237 times and the outer repeats 66 times, to reach the targeted amount of cycles (200,000)

$$T_{clk} = \frac{1}{10 \cdot 10^6} \Rightarrow \frac{1}{10 \cdot 10^6} = \frac{0.02}{cycles}$$

$$\boxed{cycles = 200,000}$$

The inner loop will execute 13 cycles worth of instructions 236 times and on the 237th time it will jump to the outer loop (this will use 3 cycles):

$$236 \cdot 13 + 3 = 3071 \ \ cycles$$

The outer one will loop for 66 times and will take an additional 6 cycles going into the inner loop. On the 67th time it will jump to the toggle Routine (4 cycles)

```
outer_loop:
    clr j
    cpi i, 66
    breq toggle
    inc i
    rjmp inner_loop

    inner_loop:
        cpi j, 236
        breq outer_loop
        inc j
        nop
        nop
        mulsu gpr0, gpr1
        mulsu gpr0, gpr1
        mulsu gpr0, gpr1
        rjmp inner_loop
```

So in total:

$$Instructions_{count} = (i-1)(6 + ((j-1) \cdot 12 + 3)) + 4$$

An additional 6 cycles worth of instructions are used for toggling the Pin.

$$Instructions_{count} = 65(9 + (236 \cdot 13)) + 10 = \boxed{200,015 \ \ cycles}$$

A drift of $1.5\mu s$ every 20ms is calculated

2.) **Timer Using Timer/Counter 0 and Interrupts**

Timers are used as peripherals, so the user sets them up once and then they function forever without clogging up the main programm loop. Timer/Counter 0 was chosen, with a resolution of 8 bits.

With a clock frequency of 10 MHz and a prescaler of 1024 the tick frequency is $\frac{10 \cdot 10^6}{1024} = 9765.625Hz$. Timer 0 has a TOP value of 255, so with the above tick frequency the Timer will overflow $\frac{9765.625}{255} \approx 38.30$ times a second. To get exactly 20ms the the timer need to be initialized with a starting value. The formula to calculate the start value is the following:

$$TCNT0 = 2^{T0_{res}} - \frac{f_{clk}}{prescaler \cdot f_{target}}$$

$$TCNT0 = 255 - \frac{10 \cdot 10^6}{1024 \cdot 50} \approx 60$$

So each time the timer overflows it needs to be re-initialized with the value of 60.

In order to have an Interrupt when the timer overflows, the TOIE0 (Timer Overflow Interrupt Enable) bit needs to be set in the TIMSK0 (Timer Interrupt Mask) Register:

```
ldi gpr0, (1<<TOIE0)
out TIMSK, gpr0
```

The semi-last thing to do is Enable the Timer itself. This needs to be last thing to do, becase once the TCCR0 register is written, the timer starts counting. For a prescaler of (5) 1024 both CS00 and CS02 bits need to be set.

```
ldi gpr0, (1<<CS00) | (1<<CS02)
out TCCR0, gpr0
```

The last step is to enable global interrups using the ‘sei‘ instruction and then jump over to the main loop (which could be anything) As soon as the timer overflows it sets the TOV0 (Timer Overflow) in the TIFR0 (Timer/Counter Interrupt Flag Register) and the program jumps to the ISR (Interrupt Service Routine) to handle the interrup appropriately

3.) **Differenses and Comparison**

Both methods yield a precise time events with minimal drift. However the looping method has 100% microprocessor usage, at all times. But, If someone would want to add a small push button, for example, it would require vast code changes in order to accomodate for the push button and also keep the loop responsive. Something that the timer/counter solves by being a peripheral.

In the span of 1 second (10 million cycles) there are 50 ticks, 20ms appart, however, the first method executes approximately 10 million instructions to achieve that. On the other hand, the timer method will use only the instructions in the ISR when the interrupt occurs. The ISR takes about 12 cycles to complete which means 600 cycles in total (the 'reti' instruction takes 4 cycles to complete with a PC of 16-bits), or about 45 instructions.

4.) **Notes**

(a) The stack has to be initialized in order to properly return from the interrup handler

```
ldi gpr0, high(ramend)
out sph, gpr0
ldi gpr0, low(ramend)
out spl, gpr0
```

(b) The directive '.def' will only work with a compatible assembler, the one that ships with Microchip Studio (avrasm) works, however 'avr-gcc' doesn't. Be mindful when using it.

(c) While not applicaple in this case, when entering an ISR, the Status Register (SREG) should be saved or push onto the stack, and restored just before 'reti'