Interrupts, Software Interrupts and Interrupt Priority in LPC1768

The cortex-m3 core which the LPC1768 uses has an extremely flexible interrupt controller called the NVIC (Nested Vectored Interrupt Controller). The NVIC has several distinguishing features.

1. Vector table is relocatable. It can even be kept in the RAM.

```
SCB->VTOR = NEW_VECTOR_TABLE_LOCATION;
```

2. The entries in the vector table are not instructions which lead to the ISR but the address of the ISR itself.

3. Interrupt entry and exit take 12 cycles irrespective of instruction being executed (maybe there are some rare exceptions to this). The point is that interrupts are very deterministic.

4. Some of the registers are pushed onto the stack automatically before ISR code execution begins and also popped off when returning (this is done within the 12 cycles mentioned above). So, the ISR code can be pure C code with no assembly wrappers.

5. Tail - chaining. If the CPU is servicing an ISR and another interrupt (lower priority, so it doesnt pre-empt the current ISR in execution) occurs, the stack is not popped. The CPU vectors to the next interrupt in 6 cycles instead of 12 cycles.

6. When an exception takes place and the processor has started the stacking process, and if during this delay a new exception arrives with higher preemption priority, the late arrival exception will be processed first.

7. Interrupt priorities can be changed during run-time and two kinds of priorities are defined i.e. pre-emption priority and sub-priority level.

8. An interrupt will never preempt itself. This reduces the chances of stack overflow. For example, if a timer is made to generate interrupts once in 100 clock cycles and the interrupt execution lasts for 200 cycles, the first time the interrupt arrives, the ISR is fully serviced and then only will it acknowledged that another interrupt is being called for and execute the ISR again. It doesn't preempt the running ISR to call the same ISR again.

9. Explicit support for software triggered interrupts.

Now lets get to some code. We'll set up a timer to generate an interrupt once in a while and toggle the LED inside the interrupt.

```c
#include "LPC17xx.h"

int main (void)
```

```
{
    LPC_SC->PCONP |= 1 << 1; //Power up Timer 0
    LPC_SC->PCLKSEL0 |= 1 << 2; // Clock for timer = CCLK
    LPC_TIM0->MR0 = 1 << 23; // Give a value suitable for the LED blinking frequency ba
    LPC_TIM0->MCR |= 1 << 0; // Interrupt on Match0 compare
    LPC_TIM0->MCR |= 1 << 1; // Reset timer on Match 0.
    LPC_TIM0->TCR |= 1 << 1; // Manually Reset Timer0 ( forced )
    LPC_TIM0->TCR &= ~(1 << 1); // stop resetting the timer.
    NVIC_EnableIRQ(TIMER0_IRQn); // Enable timer interrupt
    LPC_TIM0->TCR |= 1 << 0; // Start timer


    LPC_SC->PCONP |= ( 1 << 15 ); // power up GPIO
    LPC_GPIO1->FIODIR |= 1 << 29; // puts P1.29 into output mode. LED is connected to P

    while(1)
    {
        //do nothing
    }
    return 0;

}
void TIMER0_IRQHandler (void)
{
    if((LPC_TIM0->IR & 0x01) == 0x01) // if MR0 interrupt
    {
        LPC_TIM0->IR |= 1 << 0; // Clear MR0 interrupt flag
        LPC_GPIO1->FIOPIN ^= 1 << 29; // Toggle the LED
    }

}
```

The code is fairly simple. We set up timer 0 to run off the CPU Clock (CCLK). Match 0 is set to 2^23 (2 power 23) and we've asked Timer 0 to be reset on Match 0 and also an interrupt to be generated when Match 0 occurs. The timer starts, counts from 0 to 2^23. At this point, match occurs. The timer is reset and the interrupt occurs. Inside the interrupt, we check for the source of the interrupt (Timer 0 can produce interrupts from many sources like Mat0 , Mat1 etc..) and then toggle the LED.

Now, since the start up code gets the chip running at 100Mhz by default,
1 tick of the timer = 1 / 100Mhz = 10 ns (nano seconds)
So (2^23 + 1) ticks = 0.08388609 seconds.
We should see the LED toggling once in 0.083 secs. That's what i'm seeing in front of me.

Caveat:

1. Bit 1 of LPC_TIM0->TCR resets the timer and holds it there even if you start the timer. Make sure to clear it. Otherwise, the timer wont tick.
2. An extra 1 is added in the calculation above since the timer starts counting from 0 and not 1. (The timer in the PWM block on the other hand counts from 1).

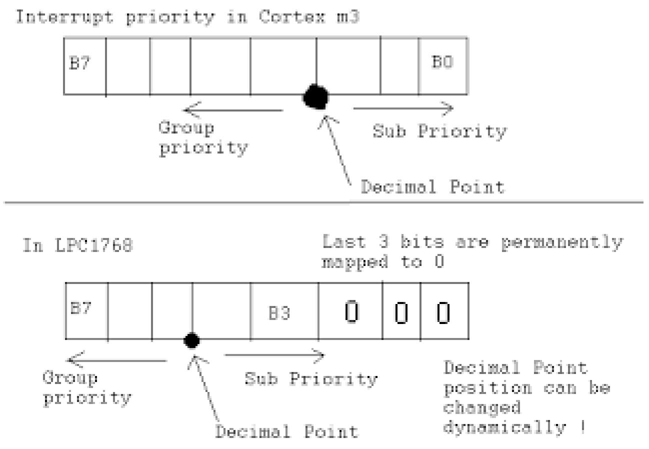
Now, we'll stop using the timer but use the timer interrupt ! This is basically a software interrupt.

```
#include "LPC17xx.h"

volatile uint32_t del;
void _delay(uint32_t delay);

int main (void)
{
  NVIC_EnableIRQ(TIMER0_IRQn);
  LPC_SC->PCONP |= ( 1 << 15 ); // power up GPIO
  LPC_GPIO1->FIODIR |= 1 << 29; // puts P1.29 into output mode. LED 
is connected to P1.29
  while(1)
  {
    _delay(1 << 24); // Wait for about 1 second
    NVIC_SetPendingIRQ(TIMER0_IRQn); // Software interrupt
  }
  return 0;

}
void TIMER0_IRQHandler (void)
{
  LPC_GPIO1->FIOPIN ^= 1 << 29; // Toggle the LED
}
void _delay(uint32_t delay)
{
  uint32_t i;
  for(i = 0;i < delay;i++ )
    del = i; // do this so that the compiler does not optimize away 
the loop.
}
```

We're accomplishing the same thing. Blinking the LED. But now, instead of using a timer, we are using a software delay. The delay is about 1 second (cant be sure as it depends on the compiler and optimization level). After the delay, the interrupt is "called". The ISR is also shorter as it doesn't have to check for the source of the interrupt within Timer 0.

And finally, we have interrupt priorities. In the LPC17xx user manual as of date, NVIC is explained in chapter 6. There we have the register description for interrupt priorities.We see that each interrupt has a 5 bit priority level associated with it. What does this 5 bit number mean ? Strangely enough, the answer is not in chapter 6 which covers NVIC.

This piece of info is presented in the appendix of the manual ! Chapter 34 . More specifically, 34.3.3.6 Interrupt priority grouping.
The 5 bit priority field for each interrupt has 2 parts separated by a decimal point. In fact, Cortex M3 provides 8 bit priority field for each interrupt. But LPC1768 has implemented only 5 bits. the 3 LSB bits are permanently mapped to 0.


Interrupt priority in Cortex m3
B7
B0
Group priority
Sub Priority
Decimal Point
In LPC1768
Last 3 bits are permanently mapped to 0
B7
B3
0
0
0
Group priority
Sub Priority
Decimal Point
Decimal Point position can be changed dynamically !

Interrupts are assigned to different groups and interrupts in each group have different priority within the group. Lets see what the implications are:

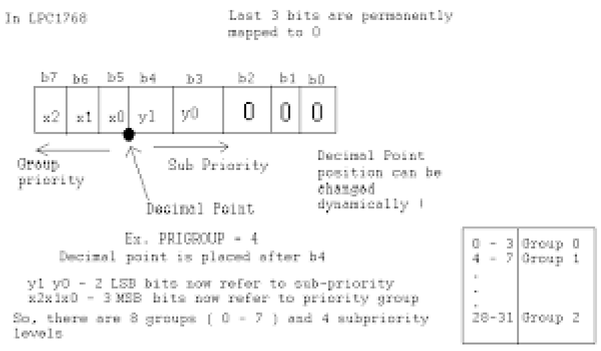
1. An interrupt belonging to a lower group (number) may preempt an interrupt of a higher group(number). Suppose Interrupt A is in group 4 and is running. Interrupt B in group 2 arrives while Int A ISR is executing. Now, interrupt A ISR is preempted and ISR for Interrupt B is executed. After ISR B is done, execution of ISR A continues.
2. If interrupt C which is also in group 4 arrives while ISR A is executing, ISR A is NOT preempted. It has to wait. After execution of ISR A, ISR C is executed.
3. If interrupt A and C arrive simultaneously, the one with lower sub-priority value is executed first.

Here is another sweet thing. The decimal point (to be accurate, binary point) demarcating group priority number from sub priority number may be positioned as the user wishes. This means we can choose and have a compromise on the number of groups vs. subpriority.

Lets see how we go about doing this. Have a look at section 34.4.3.6.1 Binary Point in the manual.

```
NVIC_SetPriorityGrouping(0x04);
```

Setting PRIGROUP to 4 puts the binary point after b4 as illustrated below. This will create 8 priority groups(3 bits for priority group) and 4 sub priority levels (2 bits for sub level) in each group.


In LPC1768
Last 3 bits are permanently mapped to 0
b7 b6 b5 b4 b3 b2 b1 b0
x2 x1 x0 y1 y0 0 0 0
Group priority
Sub Priority
Decimal Point
Decimal Point position can be changed dynamically !
Ex. PRIGROUP = 4
Decimal point is placed after b4
y1 y0 - 2 LSB bits now refer to sub-priority
x2x1x0 - 3 MSB bits now refer to priority group
So, there are 8 groups (0 - 7) and 4 subpriority levels
0 - 3 Group 0
4 - 7 Group 1
28-31 Group 2

Now, we can set the priority level for an interrupt like this

```
NVIC_SetPriority(UART3_IRQn,5);
```

This will assign UART3 IRQ to priority group 1 (see the table in the figure above) and give it a sub priority level of 1 within the group.

If you dont want non-preemptive priority levels (sub priority levels), then you can leave the decimal point at its default value (0) and just use NVIC_SetPriority. For instance , suppose we needed TIMER0_IRQ to pre-empt UART3_IRQ , then , this would work fine. This way there are 32 pre-emptive priority levels with 0 being the highest and 31 being the lowest.

```
NVIC_SetPriority(TIMER0_IRQn,3);
NVIC_SetPriority(UART3_IRQn,4); //level here has to be > level
above since UART3_IRQ can be pre-empted by TIMER0_IRQ.
```