

LPC2103 for encoders

*Up-down counter and tachometer
for optical encoders*

2008-03-18 NM



Index

1 Introduction.....	4
2 Hardware.....	5
3 Software.....	6

Pictures index

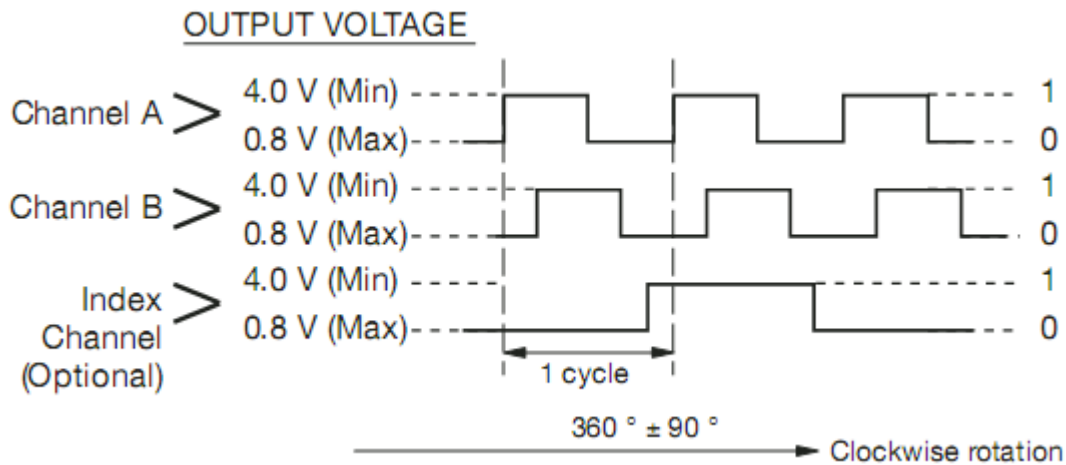
Picture 1: Encoder signals.....	4
Picture 2: System description.....	5
Picture 3: Software interrupts.....	6

***Abstract:** The purpose of this application note is the use of NXP's μ C LPC2103 as counter for incremental optical encoders. By the use of a single μ C's timer channel and a digital input, we can obtain either the up/down count, with the help of encoder's quadrature channels, either the rotation speed evaluation.*

1 Introduction

In many applications for motor control (either they are position control applications, or speed control or acceleration control) it is mandatory to have a closed control loop, with more or less complex filters (PID filters) for an optimal regulation.

The most used architecture to control movements is based on incremental optical encoders linked to the mechanical device, with a dual channel output having square wave quadrature signals (Channel A and Channel B), in order to provide at the same time the count and the rotation direction.



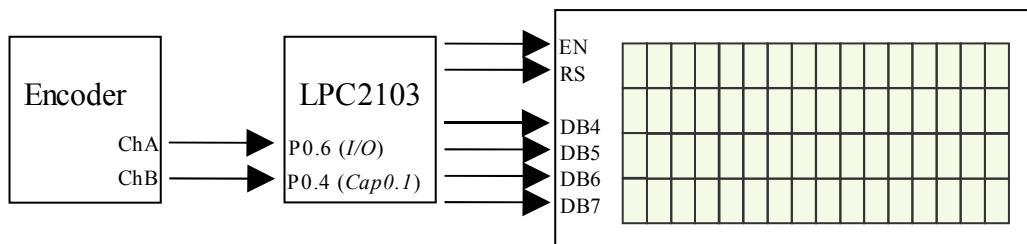
Picture 1: Encoder signals

By inspecting the Picture 1 you can view that if Channel A leads by 90° the Channel B, it is a clockwise rotation, at the opposite (if Channel B leads by 90° the Channel A) you have a counterclockwise rotation. Some encoders have an extra optional channel too (Index) signaling the complete rotation (i.e. a single pulse each 360° rotation). Channels A and B, vice versa, have pulses per revolution depending on encoder's resolution. The goal of this application note is to count pulses emitted by the encoder, checking at the same time the phase angle between the two channels in order to have an up/down counter. Moreover, by measuring time interval between two successive pulses, you can obtain rotation speed value expressed in pulse/sec.

2 Hardware

The encoder being used is a device by Bourns, with 100 pulses per revolution (allowing an angular resolution of $360^\circ/100$, that is 3.6°).

The μC is NXP's LPC2103, based on ARM7-TDMI core. We used μC 's Timer0, that has 32 bit hardware registers and is capable of recording counts from position 0 up to position $2^{32}-1$ (4,294,967,295); in most cases that's enough for a common position control. Specifically, channel B and channel A have been linked to *Timer0's* CAP0.1 pin and P0.6 I/O pin. A standard HD44780 4x16 LCD has been used to show rotation speed and position count; this device has been connected with a four bit data bus and two control lines (see Picture 2).

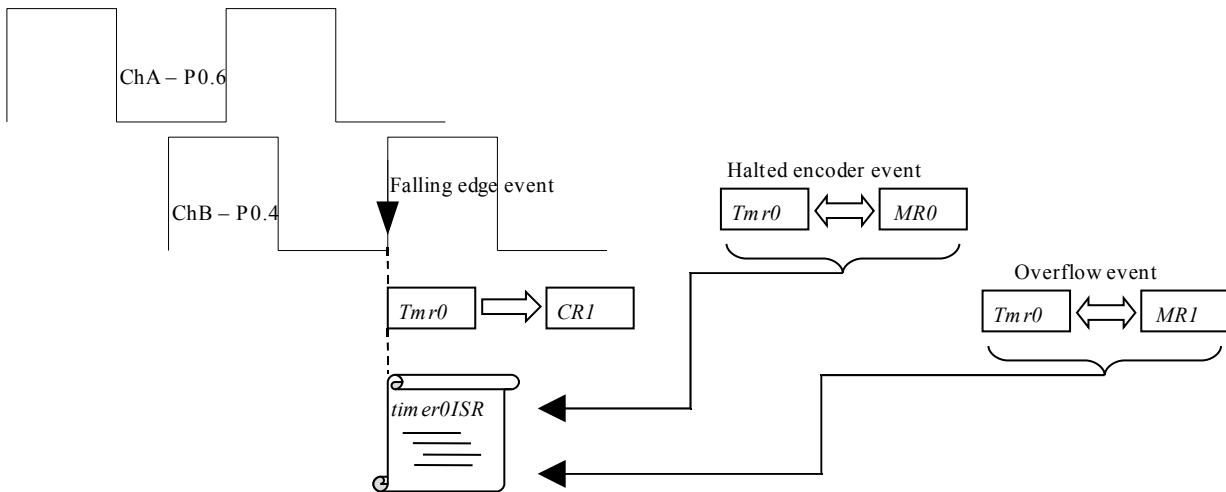


Picture 2: System description

3 Software

μ C registers have been set as follows: *Timer0* has been set in *Timer Mode* (default selection at reset), *Timer0 CAP0.1* channel has been connected to *P0.4* input and the capture action has been triggered with input falling edge. In this manner, current *Timer0* value is loaded into *CR1* register and an interrupt signals the occurring event.

Furthermore, two more interrupts have been activated, to indicate that *Timer0* has reached values preloaded into *Match Register MR0* e *MR1*; these are respectively used to update encoder speed, in zero speed condition, and to signal the *Timer0* overflow condition.



Picture 3: Software interrupts

```

PINSEL0 |= (1<<9); // set P0.4 as CAP0.1 (Timer0)

T0CCR |= (1<<4); // capture on CAP0.1 falling edge CR1 = TC
T0CCR |= (1<<5); // interrupt on CAP0.1 event

T0MCR |= 1; // interrupt on MR0 (when MR0 matches TC)
T0MCR |= (1<<3); // interrupt on MR1 (when MR1 matches TC)

T0MR0 = SPEED_UPDT_INT; // MR0 is loaded with FOSC
T0MR1 = 0xFFFFFFFF; // MR1 is loaded with max int val (overflow condition)

enIRQ(TIMER0_VIC_Ch, timer0ISR, 15); // Timer0 IRQ configuration

T0TCR = TCR_CNT_ENBL; // enable Timer0

```

Timer0 interrupt source has been connected to *VIC (Vectored Interrupt Controller)* 15 channel, with a priority level 15 and interrupt service routine *timer0ISR*.

```
void timer0ISR(void)
{
    unsigned int regValue;

    regValue = TIMER0_IR;           // read Timer0 interrupt register
    if(regValue & CR1_INT_MASK)     // CR1 interrupt
    {
        T0TCR |= TCR_CNT_RST;      // reset
        T0TCR &= ~TCR_CNT_RST;     // timer
        encSpeed = SPEED_CONST/T0CR1; // encoder speed evaluation
        T0MR0 = SPEED_UPDT_INT;    // MR0 is re-loaded with SPEED_UPDT_INT

        // read P0.6 (ChA) to compare with CAP 0.1 (ChB) (now ChB is low)
        if(IOPIN & 0x40)           // ChA is high
            encPos--;              // ChB leads ChA (counterclockwise)
        else                        // ChA is low
            encPos++;              // ChA leads ChB (clockwise)
        TIMER0_IR = regValue;      // reset Timer0 interrupt
    }
    else if((regValue&MR0_INT_MASK) && !(regValue&MR1_INT_MASK)) // MR0 interrupt (refresh time)
    {
        T0MR0 += SPEED_UPDT_INT;   // MR0 is added with SPEED_UPDT_INT
        encSpeed = SPEED_CONST/T0TC; // encoder speed evaluation (can't use T0CR1)
        TIMER0_IR = regValue;      // reset Timer0 interrupt
    }
    else                            // MR1 interrupt (overflow condition)
        encSpeed = 0;              // speed is zero

    VICVectAddr = 0x00000000;      //dummy write to VIC to signal end of interrupt
}
```

The interrupt service routine has to manage three possible causes that may trigger the event: a falling edge on *CAP0.1* and a consequent *Timer0* value loading into *CR1*; *Timer0* reaching the value in *Match Register MR0* and *Timer0* reaching the value in *Match Register MR1*.

- a) **CAP0.1 falling edge:** the encoder is rotating and (since we are in the service routine) current *Timer0* value has already been loaded into *CR1*. At this time, we can reset *Timer0* and measure the rotation speed in pulse/sec by doing the ratio between the speed constant (internal clock frequency) and the *Timer0* value saved in *CR1*. *MR0* register is reloaded with default value for the speed update in case of stopped encoder. Later we have to update up/down counter variable *encPos*, by evaluating the phase angle between ChA e ChB; as we are in the service routine, ChB is surely low (falling edge triggered), then it's enough to check current ChA level to understand if we have clockwise rotation (ChA low) or counterclockwise rotation (ChA high). In the former case we increase the 32bit *encPos* variable, in the latter we decrease it. Finally, the interrupt flag is cleared.
- b) **Timer0 matches MR0:** in this case the encoder came to a halt for a time interval

proportional to the value loaded in MR0 (in our case, one-tenth of second) and we are not in overflow. We add the value in MR0 and an extra time interval for the successive speed update, then we evaluate speed rotation expressed in pulse/sec, as the ratio between speed constant (internal clock frequency) and the current *Timer0* value. *encPos* variable is not modified, nevertheless the interrupt flag is cleared.

- c) ***Timer0* matches *MR1***: in this case the encoder came to a halt for a time interval proportional to the value loaded in MR1 (in our case $2^{32}-1$, about 5 min) and we are in overflow. The speed is set at zero and the interrupt flag is not cleared.

In all preceding cases, however, it is mandatory to clear VIC (*Vectored Interrupt Controller*) register, with a dummy write, to signal the end of interrupt service routine.