Name ___________________________________

Lab Section ______________________________

PIC – Periodic Pulse with Timer and Interrupt

Lab 4

Introduction: In last week's lab you controlled the blinking rate (on/off cycle) of an LED using a time wasting loop. The lab was a good way to get started working with the PIC without learning about the microcontroller's peripherals. However the "time wasting" method is an inefficient use of the PIC's resources. In this week's lab you will solve a similar problem using one of the microcontroller's built in timer modules. Your task is to create a short periodic pulse by utilizing the microcontroller's timer and interrupt structure.

Lab Requirements:

1. Demonstration of a periodic pulse of a width equal to the last digit of your RedID in uS. If your RedID ends in "0", make a 10uS pulse. The pulse should repeat at a 10mS (±0.1mS) interval.

Demo Check (JK)______

About the TIMER 0 Module:
The Timer 0 (TMR0) module provides a useful method of tracking time without creating inefficient time wasting loops. TMR0 can operate as either an 8-bit timer/counter with a programmable period or as 16-bit timer/counter. The Prescaler provides a method for slowing the timer count rate to give longer overflow or match cycles. For instance, if we configure TMR0 as an 8-bit timer without the Prescaler, the timer would overflow at the following rate.

$$^{4}/_{F_{OSC}} \times 2^{8} = T_{OVERFLOW}$$

When running the 16F18324 at 4MHz an 8-bit overflow cycle of TMR0 would yield a 256uS delay as follows:

$$^{4}/_{4MHz} \times 2^{8} = 256uS$$

This is overflow cycle may be too frequent for many useful applications such as creating a basic system tick scheduler. In order to consume longer durations of time we could use the timer in 16-bit mode or utilize the timer prescaler or postscaler.

There are several ways to configure TMR0 to generate an interrupt a desired interval. One method is to run the timer in 8-bit mode using the prescaler and a programmed period value so the Timer 0 Interrupt Flag (T0IF) is asserted on a match. To get started study the 8-bit Timer 0 block diagram shown below:

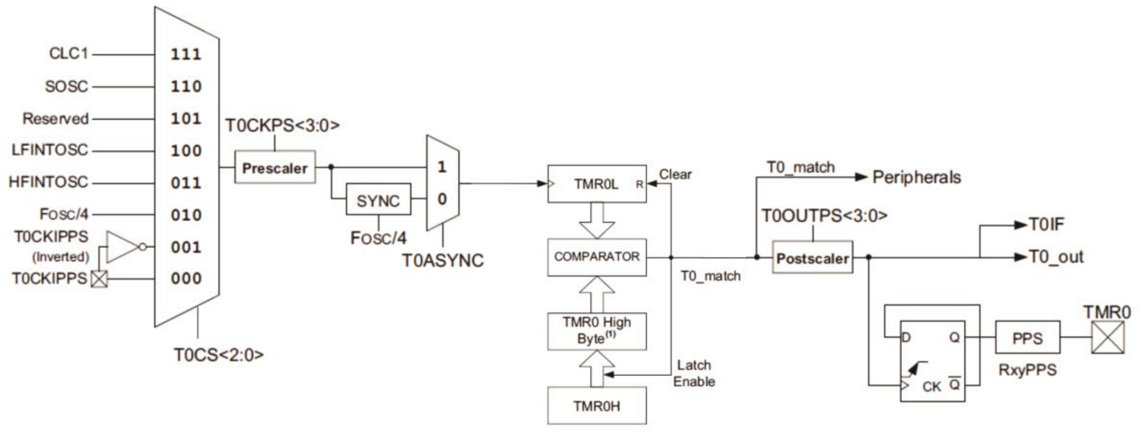


Figure 1 Block Diagram of TMR0 8-bit mode

The input to the timer/counter can be selected by setting the Timer 0 Clock Source bits (T0CS<2:0>) in the T0CON1 register. For today's lab I sugest you configure the clock rate for Fosc/4. Next you might want to slow the clock down by using the prescaler.

$$^{4}/_{F_{OSC}} \times Prescaler\ Ratio \times\ 2^8 = T_{OVERFLOW}$$

If you select a prescaler ratio of 1:64 with a 4MHz clk in 8-bit mode an overflow will happen every:

$$^{4}/_{4MHz} \times 64 \times 2^8 = 16.384mS$$

To get closer to the 10mS interval we can reduce the size of the terminal count by writing a match value to the TMR0H register. When the TMR0 count is equal to the TMR0H register the output will be asserted for one cycle and a new cycle will begin.

$$^{4}/_{4MHz} \times 64 \times (? + 1) \approx 10mS$$

For detailed information about the registers associated with TMR0 see section 25 of the PIC16f18324 datasheet.

About Interrupts:
The usefulness of the timer modules in a microcontroller is greatly enhanced by its ability to force an interrupt when the count overflow or match occurs. To enable interrupts for a TMR0 event, set the TMR0 overflow interrupt enable bit (TMR0IE) in the PIE0 register and the Peripheral Interrupt Enable (PEIE) and Global Interrupt Enable (GIE) bits in the INTCON register.

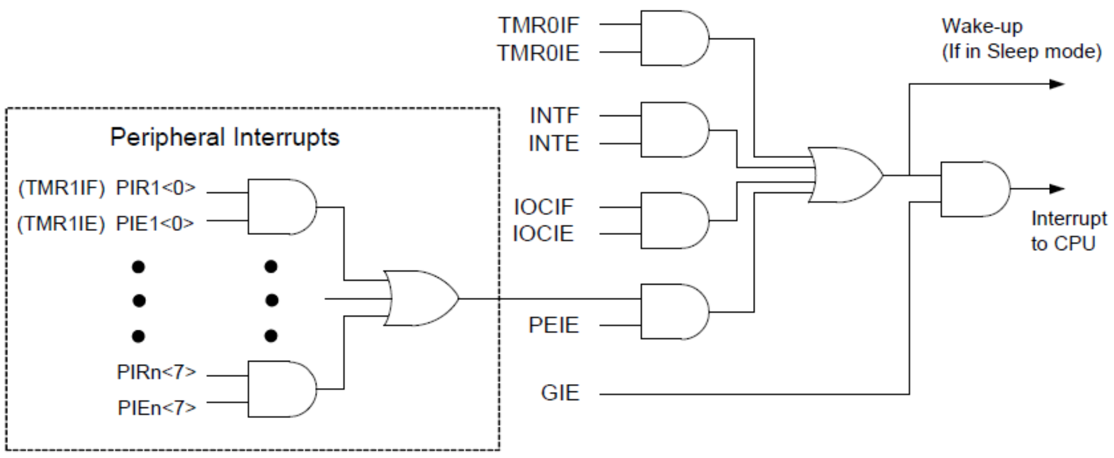


Figure 2 Interrupt Logic

Interrupts on a Mid-Range 8-bit PIC microcontroller are very basic due to their single interrupt vector address. To specify an interrupt using the XC8 compiler you create a function using the ***interrupt*** qualifier. If you are using multiple interrupt sources it is essential that you test for what caused the interrupt by performing interrupt source checks.

```
void interrupt my_isr (void)
{
  If (TMR0IF && TMR0IE)    // Source Check for TMR0
  {
    TMR0IF = 0;            // Clear TMR0 Flag

    // Put my interrupt service code here

  }
}
```

To shape the pulse you can use the NOP(); macro or the _delay(); function to make short delays at a rate of Fosc/4. Some short delays may not be achieable using “C” and you may need to write a little inline assembly.

In “C”	In “Assembly”
LATC5 = 1;	#asm
NOP();	bsf LATC,5
LATC5 = 0;	nop
	bcf LATC,5
	#endasm

Registers:

Here are some registers you may be working with today. Open the Datasheet!

REGISTER 25-3: T0CON0: TIMER0 CONTROL REGISTER 0

R/W-0/0	U-0	R-0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
T0EN	—	T0OUT	T016BIT	T0OUTPS<3:0>			
bit 7							bit 0

REGISTER 25-4: T0CON1: TIMER0 CONTROL REGISTER 1

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0
T0CS<2:0>			T0ASYNC	T0CKPS<3:0>			
bit 7							bit 0

REGISTER 25-2: TMR0H: TIMER0 PERIOD REGISTER

R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
TMR0H<7:0> or TMR0<15:8>							
bit 7							bit 0

REGISTER 7-1: INTCON: INTERRUPT CONTROL REGISTER

R/W-0/0	R/W-0/0	U-0	U-0	U-0	U-0	U-0	R-1/1
GIE	PEIE	—	—	—	—	—	INTEDG
bit 7							bit 0

REGISTER 7-2: PIE0: PERIPHERAL INTERRUPT ENABLE REGISTER 0

U-0	U-0	R/W/HS-0/0	R-0	U-0	U-0	U-0	R/W/HS-0/0
—	—	TMR0IE	IOCIE	—	—	—	INTE
bit 7							bit 0

REGISTER 7-7: PIR0: PERIPHERAL INTERRUPT STATUS REGISTER 0

U-0	U-0	R/W/HS-0/0	R-0	U-0	U-0	U-0	R/W/HS-0/0
—	—	TMR0IF	IOCIF	—	—	—	INTF[1]
bit 7							bit 0

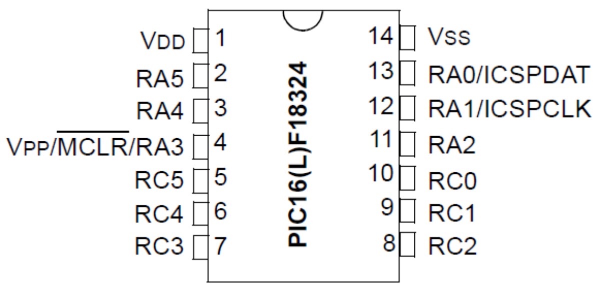
REGISTER 11-18: TRISC: PORTC TRI-STATE REGISTER

R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1	R/W-1/1
TRISC7[1]	TRISC6[1]	TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0
bit 7							bit 0

REGISTER 11-19: LATC: PORTC DATA LATCH REGISTER

R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u	R/W-x/u
LATC7[1]	LATC6[1]	LATC5	LATC4	LATC3	LATC2	LATC1	LATC0
bit 7							bit 0

PIC16F18324 Pinout:


VDD 1
RA5 2
RA4 3
VPP/MCLR/RA3 4
RC5 5
RC4 6
RC3 7
PIC16(L)F18324
14 Vss
13 RA0/ICSPDAT
12 RA1/ICSPCLK
11 RA2
10 RC0
9 RC1
8 RC2