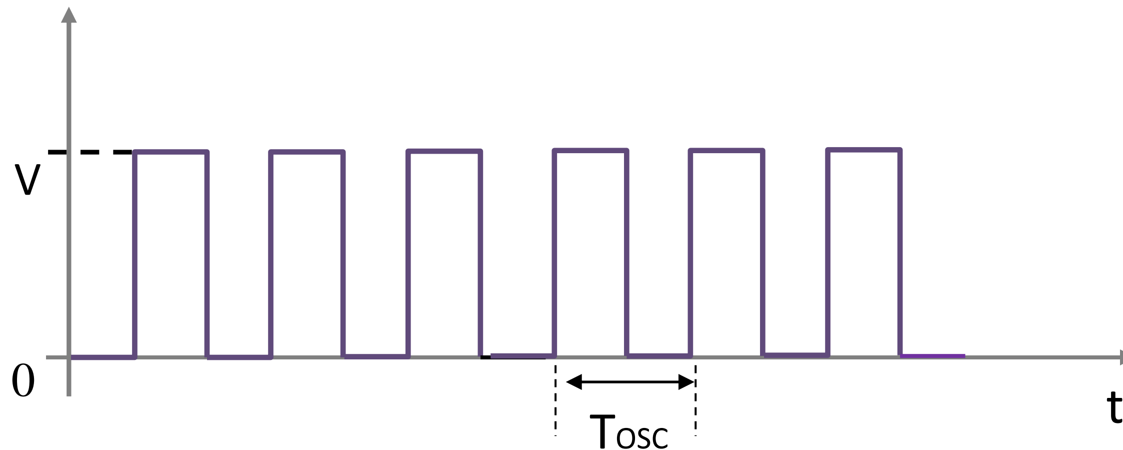
The MCU's Pulse

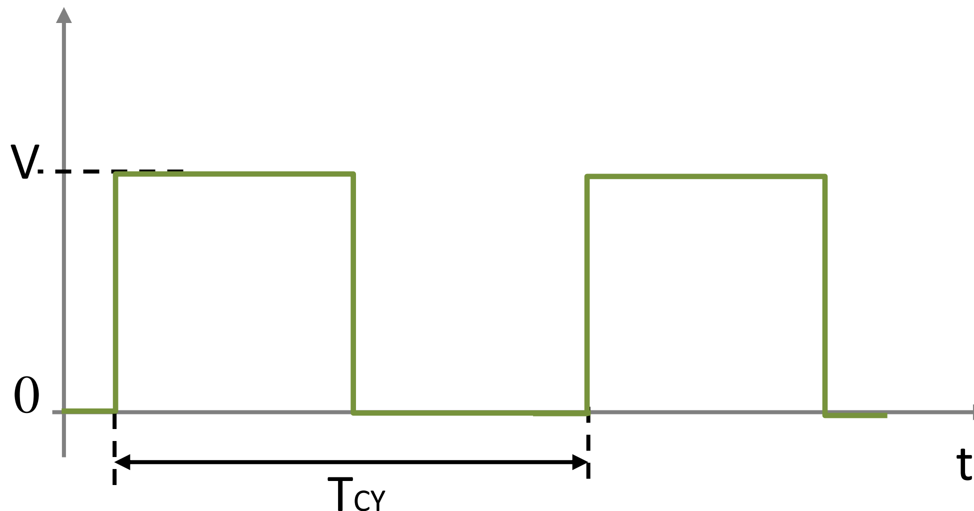
- Internal clock or oscillator to synchronize operation



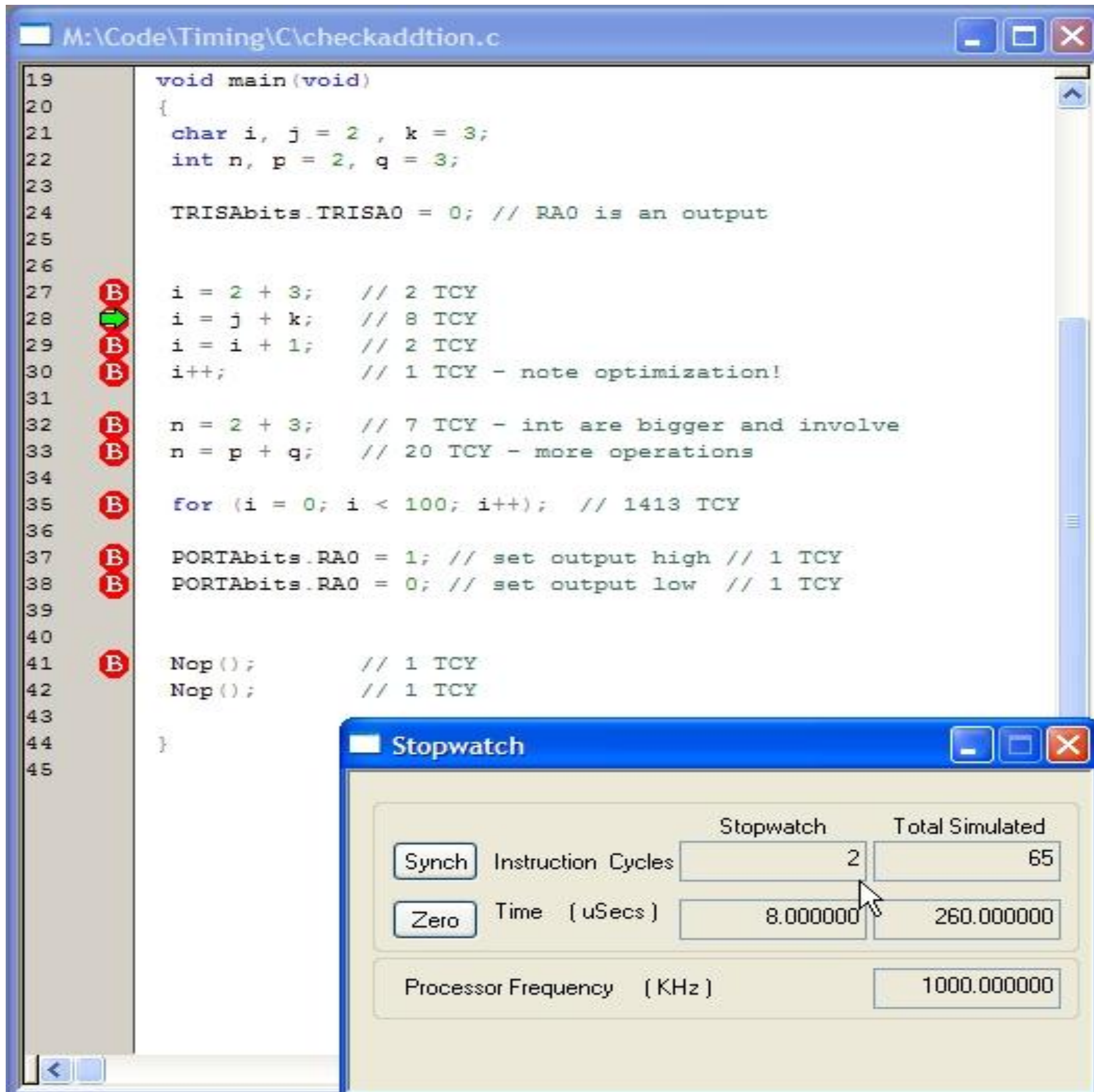
- One clock cycle = 1 T_{OSC} = 1/f_{OSC}

Clock Cycle

- The minimum time to perform any operation is one instruction cycle T_{CY}
- $1\ T_{CY} = 4\ T_{OSC}$



- Different operations require different amounts of time to complete


M:\Code\Timing\C\checkaddtion.c
void main(void)
{
char i, j = 2 , k = 3;
int n, p = 2, q = 3;
TRISAbits.TRISA0 = 0; // RA0 is an output
i = 2 + 3; // 2 TCY
i = j + k; // 8 TCY
i = i + 1; // 2 TCY
i++; // 1 TCY - note optimization!
n = 2 + 3; // 7 TCY - int are bigger and involve
n = p + q; // 20 TCY - more operations
for (i = 0; i < 100; i++); // 1413 TCY
PORTAbits.RA0 = 1; // set output high // 1 TCY
PORTAbits.RA0 = 0; // set output low // 1 TCY
Nop(); // 1 TCY
Nop(); // 1 TCY
}
Stopwatch
Stopwatch
Total Simulated
Synch
Instruction Cycles
2
65
Zero
Time (uSecs)
8.000000
260.000000
Processor Frequency (KHz)
1000.000000

Can Check T_{CY} Physically

- If configured correctly, pin 14 outputs a continuous on/off signal or square wave with period of 1 T_{CY}
- To configure pin

```
#pragma config OSC = INTIO7     // puts fosc/4 on pin 14 to
                                // check freq
```

- Pin 14 not available for other purposes (no RA6)

Can Control MCU Frequency

- SFRs: OSCCON and OSCTUNE
- Given osc.h and osc.c and set_osc_32MHz()
- OSCCON give speeds from 31KHz to 8 MHz
- OSCTUNE can multiply certain higher speeds (4 & 8 MHz) by 4 times to (16 & 32 MHz).

OSC.C

```
#include "osc.h"

void set_osc_32MHz(void)
{
    int i;

    OSCCONbits.IRCF2 = 1;  // Set the OSCILLATOR Control Register to 8 MHz
    OSCCONbits.IRCF1 = 1;
    OSCCONbits.IRCF0 = 1;
    OSCTUNEbits.PLLEN = 1; // Enable PLL, boosts speed by 4x to 32 MHz
                           // NB available for  4 & 8 MHz base only
                           // no effect otherwise
    for(i=0;i<500;i++);    // delay to allow clock PLL to lock (stabilize)
}
```

REGISTER 2-2: OSCCON: OSCILLATOR CONTROL REGISTER

R/W-0	R/W-1	R/W-0	R/W-0	R[(1)]	R-0	R/W-0	R/W-0
IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0
bit 7							bit 0

bit 7 **IDLEN:** Idle Enable bit

1 = Device enters Idle mode on SLEEP instruction
0 = Device enters Sleep mode on SLEEP instruction

bit 6-4 **IRCF2:IRCF0:** Internal Oscillator Frequency Select bits

111 = 8 MHz (INTOSC drives clock directly)
110 = 4 MHz
101 = 2 MHz
100 = 1 MHz[(3)]
011 = 500 kHz
010 = 250 kHz
001 = 125 kHz
000 = 31 kHz (from either INTOSC/256 or INTRC directly)[(2)]

bit 3 **OSTS:** Oscillator Start-up Time-out Status bit[(1)]

1 = Oscillator start-up time-out timer has expired; primary oscillator is running
0 = Oscillator start-up time-out timer is running; primary oscillator is not ready

bit 2 **IOFS:** INTOSC Frequency Stable bit

1 = INTOSC frequency is stable
0 = INTOSC frequency is not stable

bit 1-0 **SCS1:SCS0:** System Clock Select bits

1x = Internal oscillator block
01 = Secondary (Timer1) oscillator
00 = Primary oscillator

Note 1: Reset state depends on state of the IESO configuration bit.
2: Source selected by the INTSRC bit (OSCTUNE<7>), see text.
3: Default output frequency of INTOSC on Reset.

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

OSC.C

```
#include "osc.h"

void set_osc_32MHz(void)
{
    int i;

    OSCCONbits.IRCF2 = 1;  // Set the OSCILLATOR Control Register to 8 MHz
    OSCCONbits.IRCF1 = 1;
    OSCCONbits.IRCF0 = 1;
    OSCTUNEbits.PLLEN = 1; // Enable PLL, boosts speed by 4x to 32 MHz
                           // NB available for 4 & 8 MHz base only
                           // no effect otherwise
    for(i=0;i<500;i++);    // delay to allow clock PLL to lock (stabilize)
}
```

f_{osc} (MHz)	T_{osc} (ns)	T_{CY} (ns)
32		
16		
8		
4		
2		
1		
0.500		
0.250		
0.125		

f_{OSC} (MHz)	T_{OSC} (ns)	T_{CY} (ns)
32	31.25	125
16	62.50	250
8	125	500
4	250	1 000
2	500	2 000
1	1 000	4 000
0.500	2 000	8 000
0.250	4 000	16 000
0.125	8 000	32 000

Note

Although you can change the operating frequency of the MCU, f_{OSC}, (almost) every operation takes the same number to T_{CY} to complete.

For this reason operation times are always given in T_{CY} and time graphs are plotted against time in T_{CY}.

Delays

- delays.h

```
void Delay1TCY(void)                          // 1 TCY delay – same as Nop();
void Delay10TCYx(unsigned char unit)   // delay = 10 * unit * TCY
void Delay100TCYx(unsigned char unit) // delay = 100 * unit * TCY
void Delay1KTCYx(unsigned char unit)  // delay = 1000 * unit * TCY
void Delay10KTCYx(unsigned char unit) // delay = 10,000 * unit * TCY
```

- unit 1 to 255 (0 = 256)
- See C18 C Compiler Libraries PDF for details

fosc (MHz)	Delay10KTCYx(0)
32	
16	
8	
4	
2	
1	
0.500	
0.250	
0.125	

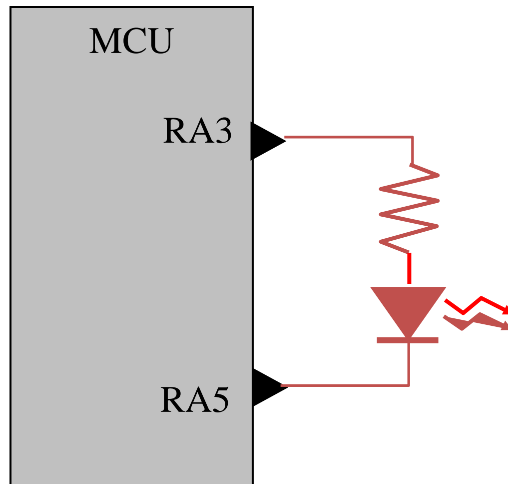
fosc (MHz)	Delay10KTCYx(0)
32	256 x 10 000 x 125 ns = 0.32 s
16	256 x 10 000 x 250 ns = 0.64 s
8	256 x 10 000 x 500 ns = 1.28 s
4	256 x 10 000 x 1 000 ns = 2.56 s
2	256 x 10 000 x 2 000 ns = 5.12 s
1	256 x 10 000 x 4 000 ns = 10.24
0.500	256 x 10 000 x 8 000 ns = 20.48 s
0.250	256 x 10 000 x 16 000 ns = 40.96 s
0.125	256 x 10 000 x 32 000 ns = 81.92 s

What is the total delay in sec?

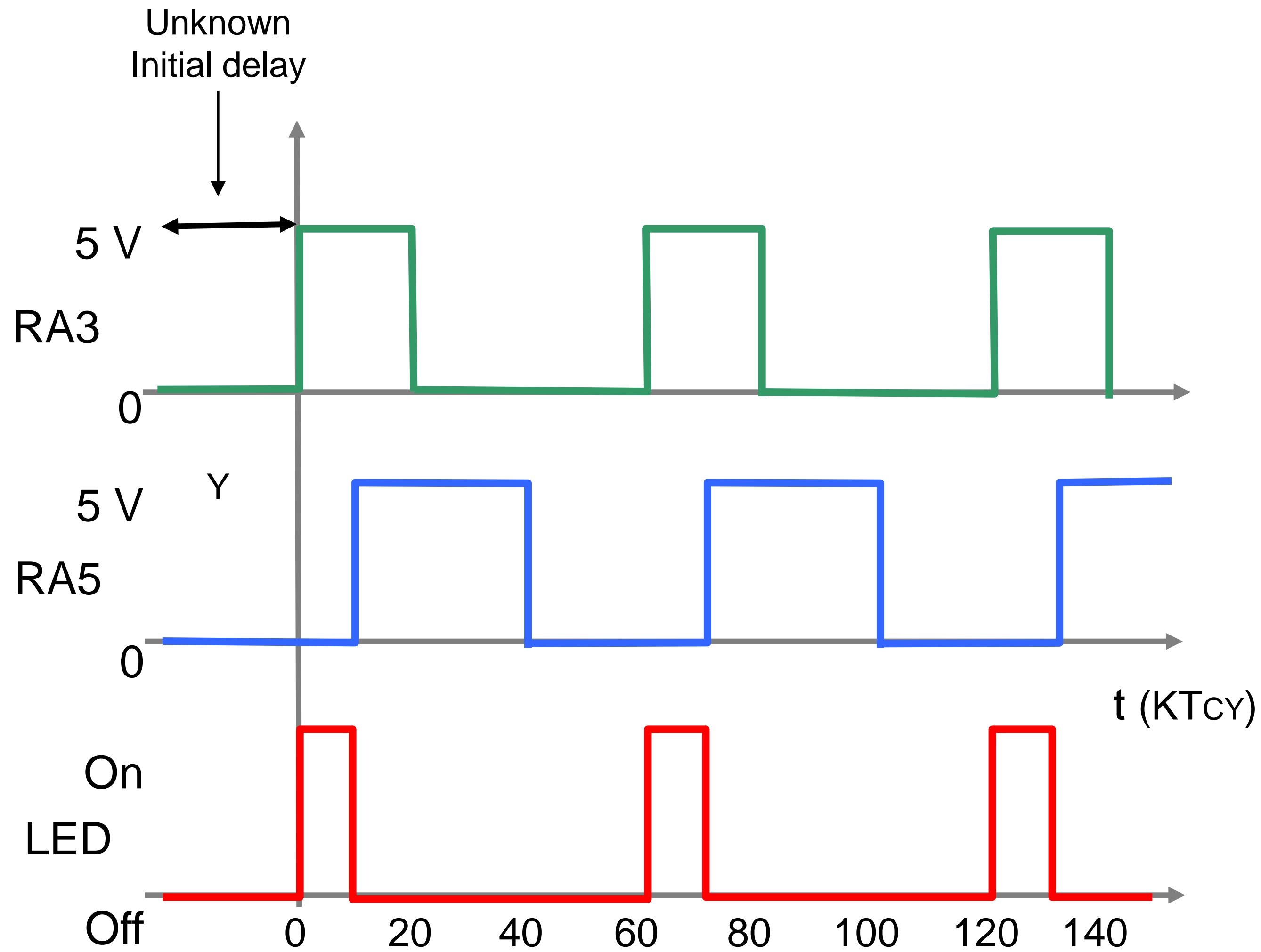
```
main()
{
set_osc_8MHz();
Delay1KTCYx(250);
Delay10TCYx(300);
while(1);
}
```

Solving Timing problems

What is the On and Off times for the LED using this circuit and code?




```
#include <delays.h>
#include "osc.h"
void main(void)
{
    Set_OSC_32MHz();
    TRISAbits.TRISA3 = 0; // set RA3 as output
    TRISAbits.TRISA5 = 0; // set RA5 as output
    PORTAbits.RA3 = 0;    // initialize to low
    PORTAbits.RA5 = 0;    // initialize to low
    while(1)              // anything in here repeats
     forever
      {
       PORTAbits.RA3 = 1;    // set RA3 high
       Delay1KTCYx(10);      // wait 1,000 * 10 TCY
       PORTAbits.RA5 = 1;    // set RA5 high
       Delay1KTCYx(10);      // wait 1,000 * 10 TCY
       PORTAbits.RA3 = 0;    // clear RA3 (low)
       Delay1KTCYx(20);      // wait 1,000 * 30 TCY
       PORTAbits.RA5 = 0;    // clear RA5 (low)
       Delay1KTCYx(20);      // wait 1,000 * 30 TCY
      }
}
```



LED Solution is drawn – not calculated

Timer Modules

- delay functions keep anything from happening.
- would like to multitask

  e.g. give 10 sec to press button

- timers are like egg timers with alarms
  - set
  - do something else
  - come back when alarm goes off

How timers work

- Two slightly different ways to use an egg timer to get a 3 minute egg
  - Set timer to 3 minutes and listen for alarm
  - Set timer to more than 3 minutes and check time every so often until 3 minutes pass
- No auditory alarm. Instead a “flag”, a one bit of memory, is set to 1

```
if ( flag == 1 ) // timer finished
```

- Counts up not down

How timers work

- Timers are counters.
- Count falling edges of internal clock which are 1 T_{CY} apart
- Can count every edge, every second edge, and so on to every N edges
- Only certain choices of N available
- Time = Count * N * T_{CY}

Timer Software

- PIC has 4 timers, will mostly use Timer0

```
#include <timers.h>
OpenTimer0(unsigned char config) ****
WriteTimer0(unsigned int timer)
ReadTimer0(void)
CloseTimer0(void)
INTCONbit.TR0IF        (flag = 0 or 1)
```

- See C18 C Compiler Libraries PDF for details

OpenTimer0

Function: Configure and enable timer0.

Include: `timers.h`

Prototype: `void OpenTimer0( unsigned char config );`

Arguments: ***config***

A bitmask that is created by performing a bitwise AND operation ('&') with a value from each of the categories listed below. These values are defined in the file `timers.h`.

Enable Timer0 Interrupt:

`TIMER_INT_ON`	Interrupt enabled
`TIMER_INT_OFF`	Interrupt disabled

Timer Width:

`T0_8BIT`	8-bit mode
`T0_16BIT`	16-bit mode

Clock Source:

`T0_SOURCE_EXT`	External clock source (I/O pin)
`T0_SOURCE_INT`	Internal clock source (TOSC)

External Clock Trigger (for `T0_SOURCE_EXT`):

`T0_EDGE_FALL`	External clock on falling edge
`T0_EDGE_RISE`	External clock on rising edge

Prescale Value:

`T0_PS_1_1`	1:1 prescale
`T0_PS_1_2`	1:2 prescale
`T0_PS_1_4`	1:4 prescale
`T0_PS_1_8`	1:8 prescale
`T0_PS_1_16`	1:16 prescale
`T0_PS_1_32`	1:32 prescale
`T0_PS_1_64`	1:64 prescale
`T0_PS_1_128`	1:128 prescale
`T0_PS_1_256`	1:256 prescale

config bitmask

Always use

- TIMER_INT_OFF,

Choices

- T0_SOURCE_INT (i.e. internal clock)
- T0_SOURCE_EXT(i.e. external edges - discuss later)
- T0_EDGE_FALL or T0_EDGE_RISE (doesn’t matter)
- T0_8BIT or T0_16BIT (usually use 16 bit counter)
- prescalers (i.e. N, count every 2^n edges)

		Time Limit (T_{CY})	
prescaler	Value	8 Bit (255)	16 Bit (65535)
T0_PS_1_1	1	0 – 255	0 – 65535
T0_PS_1_2	2	0 – 510	0 – 131070
T0_PS_1_4	4	0 – 1020	0 – 262140
T0_PS_1_8	8	0 – 2040	0 – 524280
T0_PS_1_16	16	0 – 4080	0 – 1048560
T0_PS_1_32	32	0 – 8160	0 – 2097120
T0_PS_1_64	64	0 – 16320	0 – 4194220
T0_PS_1_128	128	0 – 32640	0 – 8388480
T0_PS_1_256	256	0 – 65280	0 – 16776960

Precision (T_{CY})

prescaler	ΔT
T0_PS_1_1	1
T0_PS_1_2	2
T0_PS_1_4	4
T0_PS_1_8	8
T0_PS_1_16	16
T0_PS_1_32	32
T0_PS_1_64	64
T0_PS_1_128	128
T0_PS_1_256	256

On a 32 MHz setting, what is the longest time available? Precision?

$1 \text{ TCY} = 4 \times 1/32 \text{ MHz} = 125 \text{ ns } (125 \times 10^{-9} \text{ s})$

Largest time

$256 \times 65535 \times 125 \text{ ns} = 2.097 \text{ s}$

Precision

$256 \times 125 \text{ ns} = 0.000032 \text{ s}$

Wait for Alarm

```
#include <timers.h>

OpenTimer0(TIMER_INT_OFF & T0_SOURCE_INT &
    T0_16BIT & T0_PS_1_1);
INTCONbits.TMR0IF = 0;   // reset
WriteTimer0(15536);     // allow (65536-15536)x1 = 50K TCY
while(!INTCONbits.TMR0IF)
        { // do other things – check button or use LCD etc}
INTCONbits.TMR0IF = 0;   // reset
CloseTimer0();
```

Check Time Occasionally

```
#include <timers.h>

OpenTimer0(TIMER_INT_OFF & T0_SOURCE_INT &
    T0_16BIT & T0_PS_1_1);

WriteTimer0(0);     // allow at most 65536x1 TCY
while(ReadTimer0() <= 50000u) // only 50 000x1 TCY
        { // do other things – check button or use LCD etc}

CloseTimer0();
```

How long do you wait?

```
#include <timers.h>

OpenTimer0(TIMER_INT_OFF & T0_SOURCE_INT &
    T0_16BIT & T0_PS_1_4);
INTCONbits.TMR0IF = 0;   // reset
WriteTimer0(33555);     // ??? TCY
while(!INTCONbits.TMR0IF)
        { // do other things – check button or use LCD etc}
INTCONbits.TMR0IF = 0;   // reset
CloseTimer0();
```

- Note need prescaler for > 65536 T_{CY}

What is minimum prescaler and precision for

T_{CY}	prescaler	WriteTimer0 Start value	ΔT
1 000 215			
5 250 450			
21 000 100 004			

Blink LED 0.5s on, 0.15s off until button pressed

```
main()
{
// Configure output LED and input button pins
set_osc_32MHz();
OpenTimer0(TIMER_INT_OFF & T0_SOURCE_INT & T0_16BIT  & T0_PS_1_64);
// 0.5 s/125 ns = 4E6 TCY. 4E6/65536 = 61 – so use prescaler 64
while(1)
	{
	  WriteTimer0(0);
	  // LED on
	 while(ReadTimer0() <=  62500u && button == OFF)
		{ //monitor button}
	 WriteTimer0(0);
	  // LED off
	while(ReadTimer0() <= 18750u && button == OFF)
		{ //monitor button}
	}
}
```

Timing Events

- e.g. how long button is pressed
- Configure timer with big enough N
- Wait for start event e.g. falling edge (while loop)
- Start counter (WriteTimer0(0))
- Clear flag (INTCONbits.TM0IF = 0)
- Wait for end event e.g. rising edge (while loop)
- Time = ReadTimer() * N (in T_{CY}, may want seconds)
- Check flag for overflow (bad Time)

  if (INTCONbits.TM0IF) -> overflow

OpenTimer0() as External Counter

- e.g. count button presses automatically
- OpenTimer0() configuration
  - T0_SOURCE_EXT(only at pin 6 – RA4/T0CK1)
  - T0_EDGE_FALL or T0_EDGE_RISE
  - T0_8BIT or T0_16BIT
  - prescalers (usually N = 1)
- Must configure RA4 as input
- Button must connect to pin 6
- WriteTimer0(0) // to clear
- ReadTimer0() // keeps current count