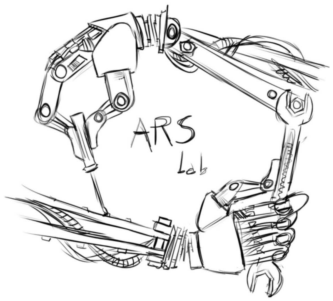
Using Timers of Microchip PIC18F Microcontrollers

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



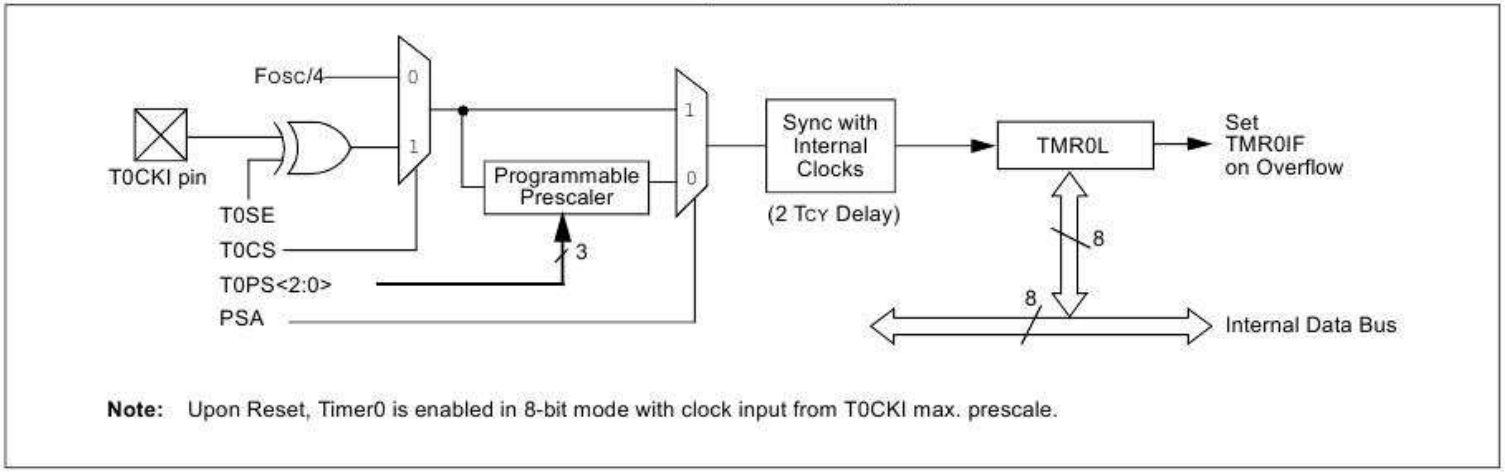
L.A.P. 1 Course

What is a “Timer”?

- It is a circuit to let a software have the “knowledge of flow of time”
- It is a composed of:
  - A **clock source**; usually the *system clock* or an *external signal*;
  - A **programmable frequency divisor**, called **prescaler**, to divide clock source frequency, if needed;
  - Some **SFRs** which hold a 8-, 16- or 32-bit value that is **incremented in hardware** using the clock source.
  - Some **SFRs** which give some *state information*, e.g **overflow** (zero crossing).
- PIC18F family has **7 timers**, called TIMER0, TIMER1, ..., TIMER5, TIMER6
- Each timer has different characteristics and may be used together with other peripherals.

The TIMER0 of PIC18

FIGURE 11-1: TIMER0 BLOCK DIAGRAM (8-BIT MODE)

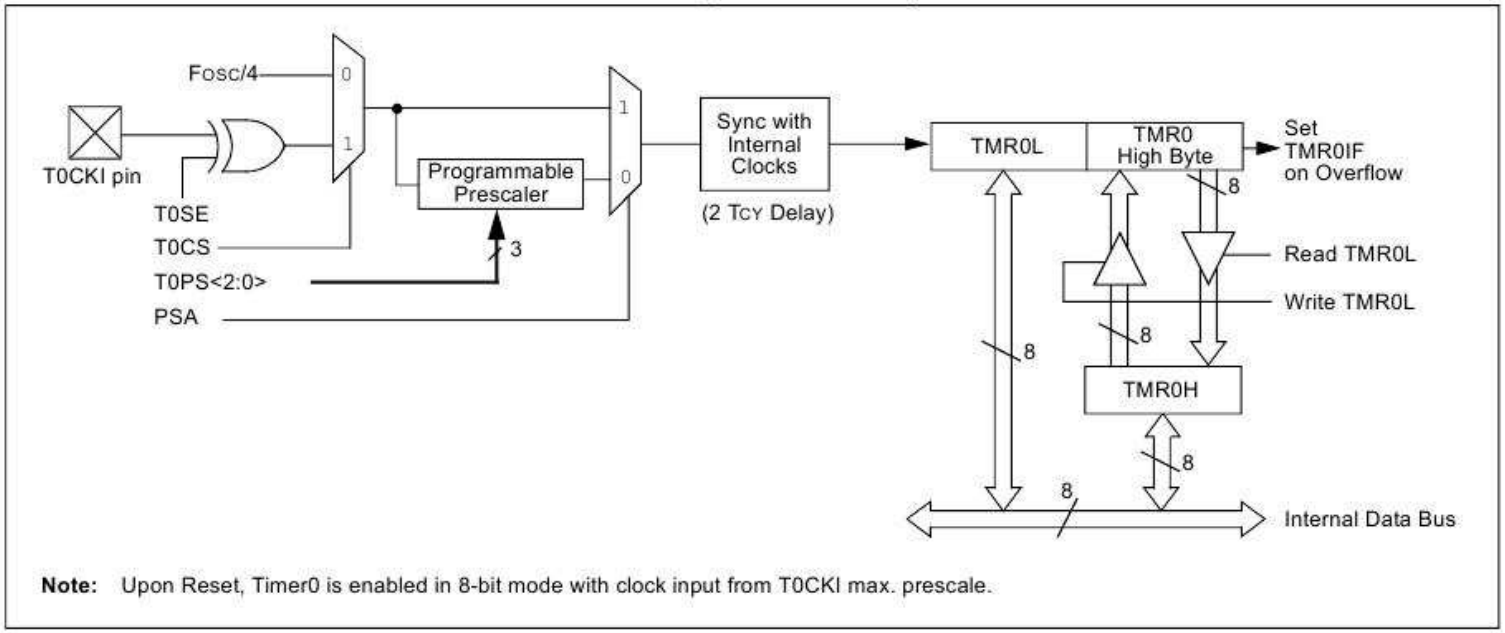


Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

- TIMER0 is a 8/16 bit timer/counter (figure shows the 8bit mode);
- **`TMR0L`** is the SFR containing the value that is incremented;
- All the parts to the left are the **clock source** circuits.
- **`T0CON`** (Timer 0 Control) register is used to program the timer, and includes the bits shown in figure (T0CS, PSA, T0PS, etc.)

The 16-bit version of TIMER0

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)

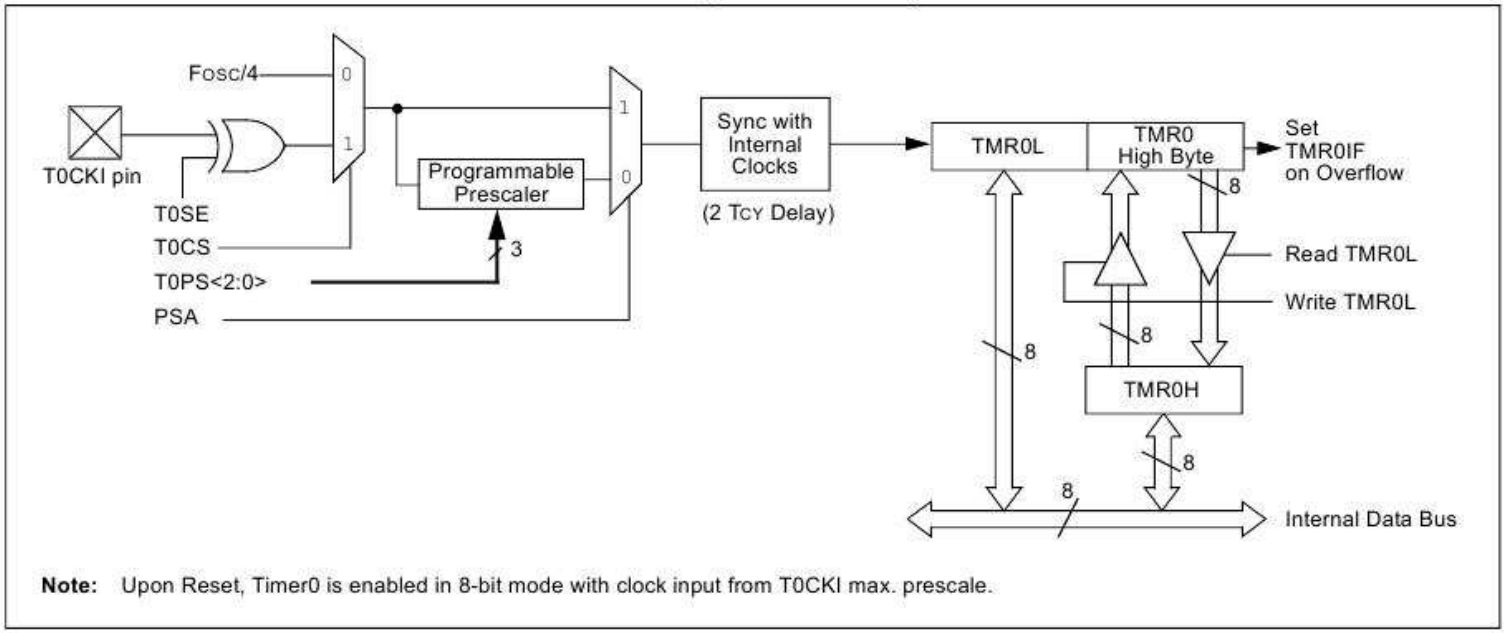


Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

- In 16-bit mode, two SFR are used **`TMR0L`** and **`TMR0H`**;
- In write operations, **`TMR0H`** must be written before **`TMR0L`**;
- In read operations, **`TMR0L`** must be read before **`TMR0H`**;
- However, XC8 offers a single 16-bit variable **`TMR0`** which includes both low and high part of TMR0.

The 16-bit version of TIMER0

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

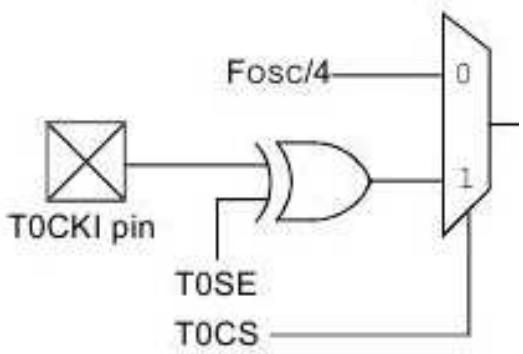
REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	TOPS<2:0>		
bit 7							bit 0

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

The **T0CON** (Timer 0 Control) SFR includes all the bits which control TIMER0 functioning.

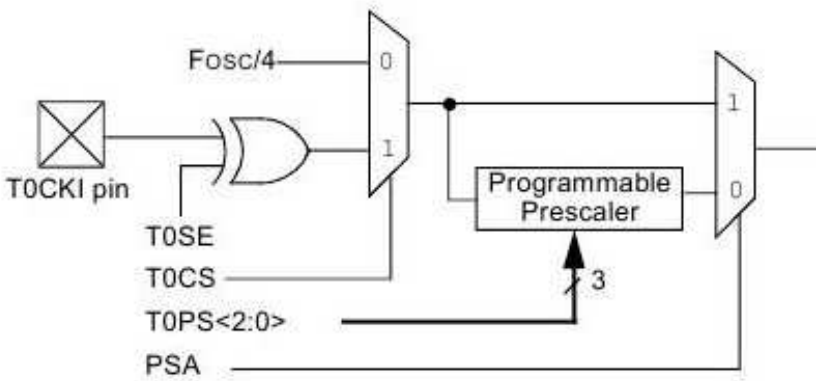
TIMER0: Selecting clock source



Clock source can be **internal** or **external** and is controlled by bit **T0CS**:

- **T0CS = 0;** $\rightarrow$ clock source is internal and is taken from $F_{osc}/4$.
- **T0CS = 1;** $\rightarrow$ clock source is external and is taken from T0CKI pin; in this case **T0SE** controls the **edge** of the signal which triggers increment.

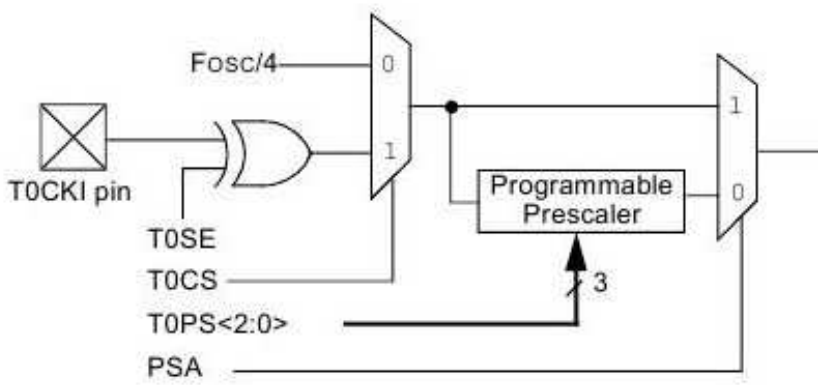
TIMER0: Dividing clock frequency



In some cases, the clock coming from the oscillator could be **too fast** for our applications: we can **lower** it by using the **frequency prescaler**.
The prescaler is a circuit which divides the signal frequency by 2, 4, 8, 16, ..., 256.
The prescaler is activated by bit **PSA**:

- **PSA = 0;** $\rightarrow$ prescaler is selected, frequency division is controlled by bits **T0PS**.
- **PSA = 1;** $\rightarrow$ prescaler is not selected.

TIMER0: Dividing clock frequency



When the prescaler is activated (**PSA = 0**), division is performed as:

- `T0PS = 111`, division 1:256
- `T0PS = 110`, division 1:128
- `T0PS = 101`, division 1:64
- ....
- `T0PS = 000`, division 1:2

TIMER0: controlling depth and on/off

REGISTER 11-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS<2:0>		
bit 7							bit 0

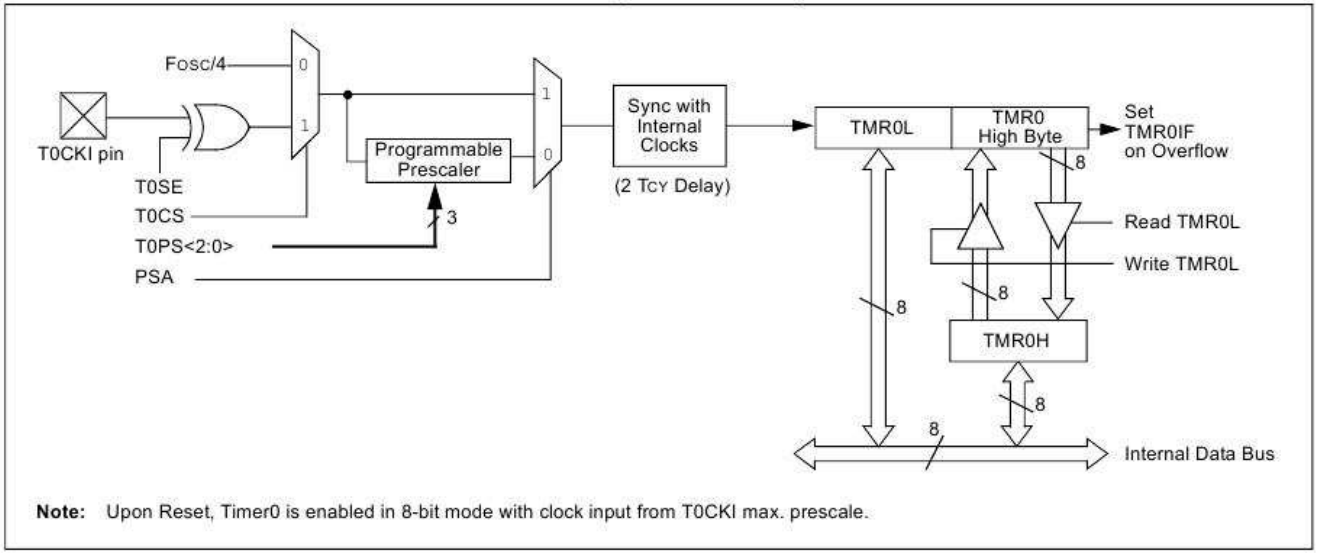
Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Finally, T0CON includes these other two bits:

- **TMR0ON**, turns on/off the timer;
- **T08BIT**, selects 8 (value "1") or 16 (value "0") bit mode.

A case-study: a timer to flash a LED

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)

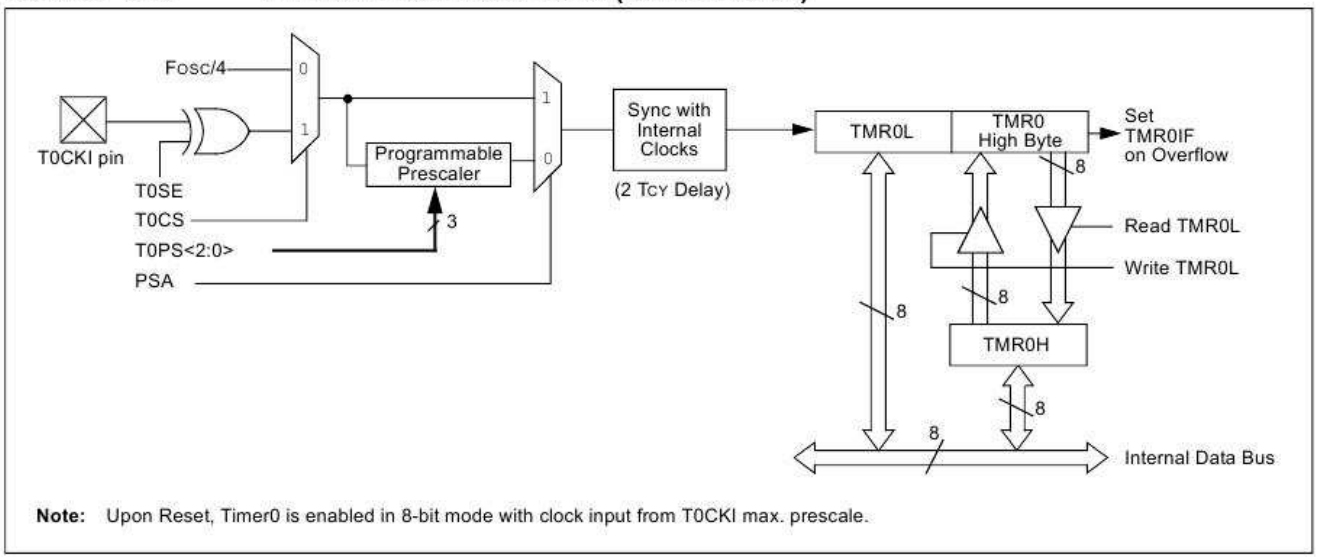


Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

- We want to use the system clock, `T0CS = 0`;
- In our board, we have $F_{OSC} = 64MHz$, therefore the basic frequency is $F_{OSC}/4 = 16MHz$, the $P = 62.5ns$;
- Let's use the prescaler and divide the frequency by 256, so `PSA = 0; T0PS = 0b111;`
- The timer increments using a period $P = 62.5ns \cdot 256 = 16\mu s$.

A case-study: a timer to flash a LED

FIGURE 11-2: TIMER0 BLOCK DIAGRAM (16-BIT MODE)



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI max. prescale.

- ... the timer increments using a period $P = 62.5ns \cdot 256 = 16\mu s$.
- Let us suppose we want a period of half a second $500ms$
- Therefore $\frac{500 \cdot 10^{-3}}{16 \cdot 10^{-6}} = 31250$
- A delay of $500ms$ implies 31250 counts

A case-study: a timer to flash a LED

```
int main(void)
{
  TRISBbits.TRISB0 = 0; // output

  T0CONbits.TMR0ON = 0; // stop the timer
  T0CONbits.T08BIT = 0; // timer configured as 16-bit
  T0CONbits.T0CS = 0; // use system clock
  T0CONbits.PSA = 0; // use prescaler
  T0CONbits.T0PS = 0b111; // prescaler 1:256 ('0b' is a prefix for binary)
  TMR0 = 0; // clear timer value
  T0CONbits.TMR0ON = 1; // start the timer

  for (;;) {
    unsigned int t;
    t = TMR0;
    if (t >= 31250) { // equivalent of 500 ms
      TMR0 = 0;
      LATBbits.LATB0 = !LATBbits.LATB0;
    }
  }
}
```

Case-study 2: more LEDs flashing

Let us suppose we want to:

- flash led in RB0 at a period of 500 ms
- flash led in RB1 at a period of 750 ms

Do we need two timers?? **NO!**

1. compute the **greatest common divisor**, which is $250ms$
2. use it as your "timer period"
3. toggle RB0 after two periods
4. toggle RB1 after three periods

Case-study 2: more LEDs flashing

- Using the same set-up of the previous example, since our period is $250ms$
- we have $\frac{250 \cdot 10^{-3}}{16 \cdot 10^{-6}} = 15625$
- A delay of $250ms$ implies 15625 counts

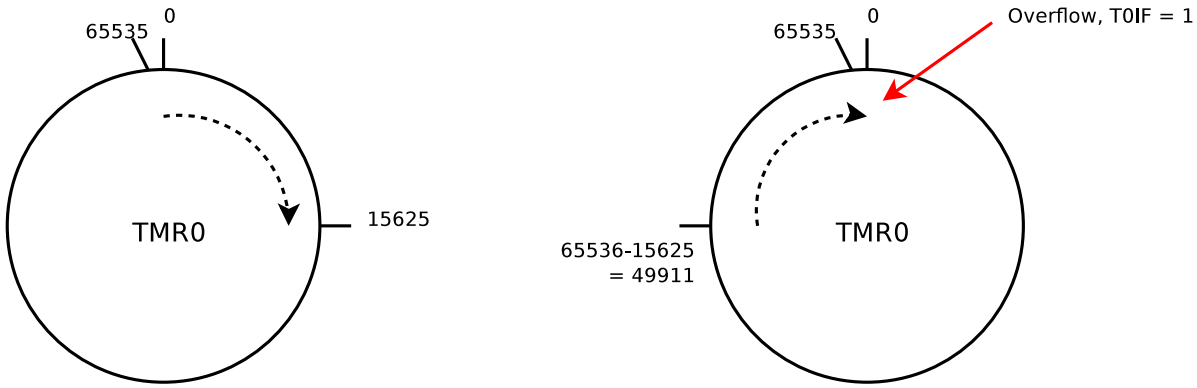
Case-study 2: more LEDs flashing

```
int main(void)
{
  char c0 = 0, c1 = 0; // why char? because they are 8 bits

  TRISBbits.TRISB0 = 0; // output
  TRISBbits.TRISB1 = 0; // output

  T0CONbits.TMR0ON = 0; // stop the timer
  T0CONbits.T08BIT = 0; // timer configured as 16-bit
  T0CONbits.T0CS = 0; // use system clock
  T0CONbits.PSA = 0; // use prescaler
  T0CONbits.T0PS = 0b111; // prescaler 1:256 ('0b' is a prefix for binary)
  TMR0 = 0; // clear timer value
  T0CONbits.TMR0ON = 1; // start the timer

  for (;;) {
    unsigned int t;
    t = TMR0;
    if (t >= 15625) { // equivalent of 250 ms
      TMR0 = 0;
      ++c0; ++c1;
      if (c0 == 2) { // flash led 0
        LATBbits.LATB0 = !LATBbits.LATB0;
        c0 = 0;
      }
      if (c1 == 3) { // flash led 1
        LATBbits.LATB1 = !LATBbits.LATB1;
        c1 = 0;
      }
    }
  }
}
```

Timer Overflow

- In our examples, we check the timer value and, after reaching a certain maximum, we clear it
- However, what does it happen if we don't modify TMR0?
- At a certain point, the TMR0 reaches its maximum possible value, which is 255 (0xff) at 8 bit and 65535 (0xffff) at 16 bit
- The next increment will **overflow** TMR0, which thus **goes to zero**
- This event is signalled by **the hardware by setting a proper bit** in a SFR
- The bit is called **T0IF** and belongs to register **INTCON**
- The bit **set by the hardware** and **cleared by software**

Timer Overflow



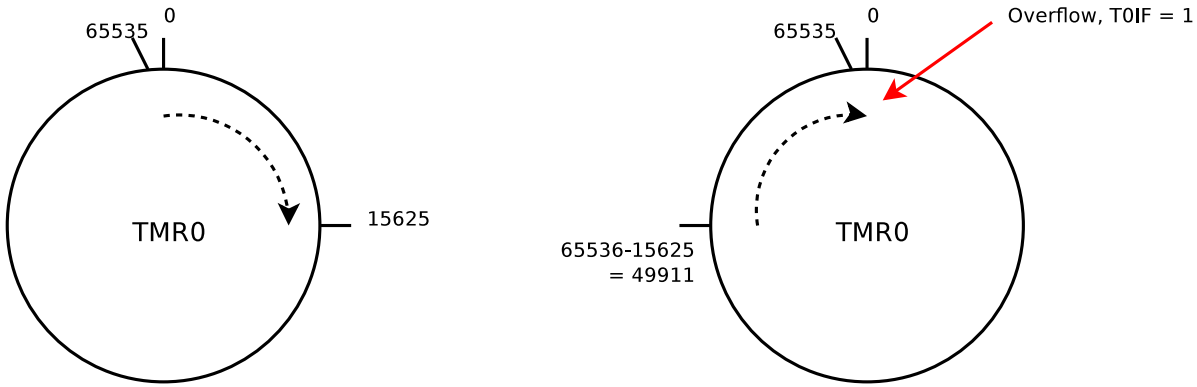
We can exploit the overflow event as follows.
Instead of clearing TMR0 and waiting for reaching our MAX (15625 in the example), we can:

- Set TMR0 to "$65536 - MAX$" ("$65536 - 15625 = 49911$" in our example)
- Wait for **overflow** by checking T0IF
- Clear T0IF

Case-study 2: LED flashing with overflow

```
int main(void)
{
  char c0 = 0, c1 = 0; // why char? because they are 8 bits

  TRISBbits.TRISB0 = 0; // output
  TRISBbits.TRISB1 = 0; // output

  T0CONbits.TMR0ON = 0; // stop the timer
  T0CONbits.T08BIT = 0; // timer configured as 16-bit
  T0CONbits.T0CS = 0; // use system clock
  T0CONbits.PSA = 0; // use prescaler
  T0CONbits.T0PS = 0b111; // prescaler 1:256 ('0b' is a prefix for binary)
  TMR0 = 49911; // initial timer value
  INTCONbits.T0IF = 0; // clear the overflow bit initially
  T0CONbits.TMR0ON = 1; // start the timer

  for (;;) {
    if (INTCONbits.T0IF == 1) { // overflow!
      TMR0 = 49911;  // reload timer
      INTCONbits.T0IF = 0; // clear overflow
      ++c0; ++c1;
      if (c0 == 2) { // flash led 0
        LATBbits.LATB0 = !LATBbits.LATB0;
        c0 = 0;
      }
      if (c1 == 3) { // flash led 1
        LATBbits.LATB1 = !LATBbits.LATB1;
        c1 = 0;
      }
    }
  }
}
```

Timer Overflow



Let's consider the expression: "$65536 - MAX$":

- We notice that 65536, in 16-bit arithmetic, **does not exist** and is **equivalent to 0**
- therefore, "$65536 - MAX = -MAX$"

Case-study 2: LED flashing with overflow

```
int main(void)
{
  char c0 = 0, c1 = 0; // why char? because they are 8 bits

  TRISBbits.TRISB0 = 0; // output
  TRISBbits.TRISB1 = 0; // output

  T0CONbits.TMR0ON = 0; // stop the timer
  T0CONbits.T08BIT = 0; // timer configured as 16-bit
  T0CONbits.T0CS = 0; // use system clock
  T0CONbits.PSA = 0; // use prescaler
  T0CONbits.T0PS = 0b111; // prescaler 1:256 ('0b' is a prefix for binary)
  TMR0 = -15625; // initial timer value
  INTCONbits.T0IF = 0; // clear the overflow bit initially
  T0CONbits.TMR0ON = 1; // start the timer

  for (;;) {
    if (INTCONbits.T0IF == 1) { // overflow!
      TMR0 = -15625;  // reload timer
      INTCONbits.T0IF = 0; // clear overflow
      ++c0; ++c1;
      if (c0 == 2) { // flash led 0
        LATBbits.LATB0 = !LATBbits.LATB0;
        c0 = 0;
      }
      if (c1 == 3) { // flash led 1
        LATBbits.LATB1 = !LATBbits.LATB1;
        c1 = 0;
      }
    }
  }
}
```

Comparing the techniques

Let's compare (1)

```
unsigned int t;
t = TMR0;
if (t >= 15625) { // equivalent of 250 ms
  TMR0 = 0;
```

to (2)

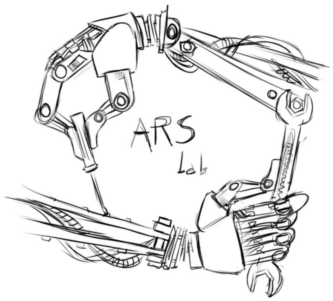
```
if (INTCONbits.T0IF == 1) { // overflow!
  TMR0 = -15625;  // reload timer
  INTCONbits.T0IF = 0; // clear overflow
```

- (1) uses a 16-bit comparison, (2) uses a single-bit comparson $\rightarrow$ less code since the CPU is 8-bit
- (2) uses polling but can be easily transformed into a **interrupt-based code** since overflows can be programmed to generate **interrupts**

Using Timers of Microchip PIC18F Microcontrollers

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



L.A.P. 1 Course