# Exercise-4: Timers

- Timers in 8051 – specifications and applications
- Calculation of timer values for the given delay
- Writing an ISR for timer overflow interrupt
- Configuring counter to count external events
- SFR's descriptions
- Using Performance analyzer feature of Keil to view delay caused by using hardware timers
- Using debugger facility to view overflow interrupt generation and ISR execution
- Using IO Port 3.4 to generate external events for counting and display the count on IO ports
- Hands On:
    - Toggling port pins using hardware timer delay
    - Using timer to generate interrupts
    - Using timer in counter mode to count the external events

- Timers
- All members of the 8051 family have at least two timer/counters, known as Timer 0 and Timer 1: most also have an additional timer (Timer 2).

- These are 16-bit timers, which mean they can hold values from 0 to 65535 (decimal).

- Timers like these are crucial to the development of embedded systems.

- To see why, you need to appreciate that, when configured appropriately, the timers are incremented periodically: specifically, in most 8051 devices, the timers are incremented every 12 oscillator cycles. (12 oscillator cycles constitute 1 machine cycle – so basically timer counts machine cycles)

- Thus, assuming we have a 12 MHz oscillator, the timer will be incremented 1 million times per second.

- There are many things we can do with such a timer:
    - We can use it to measure intervals of time.
        - For example, we can measure the duration of a function by noting the value of a timer at the beginning and end of the function call, and comparing the two results.

    - We can use it to generate precise hardware delays

    - We can use it to generate 'time out' facilities: this is a key requirement in systems with real-time constraints
    - We can use it as a baud rate generator for serial communication
    - Most important of all, we can use it to generate regular 'ticks', and drive an operating system

- In most cases, we will be concerned with 16-bit timers.

    - Assuming the count starts at 0, then – after 65.535 ms – our 12 MHz 8051 will reach its maximum value (65535) and the timer will then 'overflow'.

    - When it overflows, a hardware flag will be set. We can easily read this flag in software.

    - This is very useful behavior. For example, if we start the timer with an initial value of zero and wait until the flag is set, we know that precisely 65.535 ms will have elapsed.

    - More importantly, we can vary the initial value stored in the timer: this allows us to generate shorter, precisely-timed, delays of – say – 50ms or 1ms, as required

- Calculations of hardware delays generally take the following form:
    - Configure the timer in required mode
    - We calculate the required starting value for the timer.
    - We load this value into the timer.
    - We start the timer.
    - The timer will be incremented, without software intervention, at a rate determined by the oscillator frequency; we wait for the timer to reach its maximum value and 'roll over'.
    - The timer signals the end of the delay by changing the value of a flag variable.

- If we are using a '12-oscillations per instruction' 8051, running at 12 MHz, the longest delay that can be produced with a 16-bit timer is approximately 65 ms. If we need longer delays, we can repeat the steps above

**Case 1: OSC 12MHz**

Assuming we used a 12 MHz oscillator,
Timers get incremented every 12 osc cycles on a standard 8051
So in one second - the timer is incremented 1 million times per second.

In most cases, we will be concerned with 16-bit timers.
So the range of timer is = 0 - 65535
OSC freq = 12 MHz
1 sec = 1 million increments (1000000)
1 increment = 1/1000000 => 0.000001 or 1 microseconds
16b timer max value = 65535
65535 increments = 1/1000000 * 65535 = 0.065535 microseconds or 65.535 ms

So here, 1 sec = 1000000 increments
Since 1 sec = 1000ms
=> 1000ms = 1000000 increments
For ex: 10ms = 1000000/1000 * 10 = 10000 increments

For 10ms delay, 16b timer should start from=> 65536 – 1000 = 55536

**Case 2: OSC 11.0592MHz**

OSC freq = 11.0592 MHz
1 sec = 11.0592 MHz / 12 =>921600 increments
1 increment = 1/921600 => 1.085 us

1 sec = 921600 increments
1000 ms

10 ms = 921600 / 1000 * 10 => 9216 (full cycle)
5 ms => 9216/2 = 4608 increments (one half cycle)
65536 - 4608 => 60928 => EE00H

- How these timers operate, we need to explain the features of SFR Registers of the Timer Peripheral in AT89C51ED2 Device:
    - The TCON SFR;
    - The TMOD SFR; and,
    - The THx and TLx SFR registers.

**The TCON SFR**

We first need to introduce the TCON special function register (SFR): see table below

The various bits in the TCON SFR have the following functions:

| Bit | 7 (MSB) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (LSB) |
|---|---|---|---|---|---|---|---|---|
| NAME | TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |

TCON SFR – Note that grey bits are not connected with either timer 0 or timer 1

Some more modern 8051 devices these require fewer than 12 oscillator cycles per timer increment.

**TR0, TR1 Timer run control bits**

These bits need to be set to 1 (by the programmer) to run the corresponding timer (TR0 controls Timer 0, TR1 controls Timer 1).

If set to 0, the corresponding timer will not run.

Default value (after reset) is 0.

**TF0, TF1 Timer overflow flags**

These bits are set to 1 (by the microcontroller) when the corresponding timer overflows.

They need to be manually cleared (set to 0) by the programmer.

Default value (after reset) is 0.

**IE0, IE1 Interrupt flags**

Set by hardware when an (external) interrupt is detected on Pin 12 or Pin 13, respectively.

These features are not related to Timer 0 or Timer 1 and are not used and can be left at their default value.

**IT0, IT1 Interrupt type control bit**

Used to determine with the external interrupt flags (above) are set in response to a falling edge (‘edge triggered’) or a low-level (‘level triggered’) input on the relevant port pins.

These features are not related to Timer 0 or Timer 1 and are not used and can be left at their default value.

**What about interrupts?**

The code we use to create delays will take the following form:

```
while (TF0 == 0); // Wait until Timer 0 overflows
```

That is, we wait until the timer overflow flag indicates that the timer has reached its maximum value.

It should be pointed out that the overflow of the timers can be used to generate an interrupt: this can allow you to perform other activities while the counting goes on.

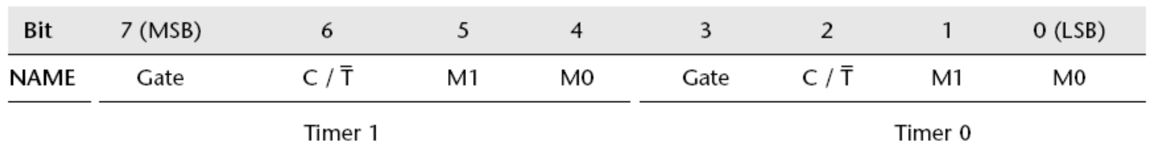
This feature is useful while creating an embedded operating system

To disable the generation of interrupts, we can use the C statements:

```
ET0 = 0; // No interupts (Timer 0)
ET1 = 0; // No interupts (Timer 1)
```

## The TMOD SFR

TMOD SFR is shown below



| Bit | 7 (MSB) | 6 | 5 | 4 | 3 | 2 | 1 | 0 (LSB) |
|---|---|---|---|---|---|---|---|---|
| NAME | Gate | C / $\overline{T}$ | M1 | M0 | Gate | C / $\overline{T}$ | M1 | M0 |
| | Timer 1 | | | | Timer 0 | | | |

The main thing to note is that there are three modes of operation (for each timer), set using the M1 and M0 bits.

We will only be concerned in this book with Mode 1 and Mode 2, which operate in the same way for both Timer 0 and Timer 1, as follows:

**Mode 0 (M1 = 0; M0 = 0) – 13b Timer Mode**

**Mode 1 (M1 = 0; M0 = 1)**
16-bit timer/counter (with manual reload) Mode

The distinction between 'automatic reload' and 'manual reload' timers has no impact on the hardware delay code

**Mode 2 (M1 = 1; M0 = 0)**
8-bit timer/counter (with 8-bit automatic reload) Mode

**Mode 2 (M1 = 1; M0 = 1) – Split Timer Mode**

The remaining bits in TMOD have the following purpose:

**GATE gating control**
Ensure that this bit is cleared, which means that Timer 0 or Timer 1 will be enabled whenever the corresponding control bit (TR0 or TR1) is set in the TCON SFR.

**C / T– Counter or timer select bit**
Ensure that this bit is cleared (for the appropriate timer), so that timer mode is used.

**The THx and TLx registers**

The two 8-bit registers associated with each timer: these are known as TL0 and TH0, and TL1 and TH1.
Here, ‘L’ and ‘H’ refer to ‘low’ and ‘high’ bytes, as will become clear shortly.

In all of the delay functions, we will use the timers in Mode 1: that is, as 16-bit timers.

If we are using Timer 1 to generate the delays, then the values loaded into TL1 and TH1 at the start of the delay routine will determine the delay duration.

For example, suppose we wish to generate a 15 ms hardware delay.
We will assume that we are using a 12 MHz 8051, and that this device requires 12 oscillator cycles to perform each timer increment:
The timer is incremented at a 1 MHz rate.

A 15 ms delay therefore requires the following number of timer increments:

$$\frac{15ms}{1000ms} \times 1000000 = 15000 \text{ Increments.}$$

15*ms* ----------------~ 1000000 = 15000 increments.
1000*ms*
The timer overflows when it is incremented from its maximum count of 65535.

Thus, the initial value we need to load to produce a 15ms delay is:
65536 – 15000 = 50536 (decimal) = 0xC568

We can load this initial value into Timer 1 as follows:
TH1 = 0xC5; // Timer 1 initial value (High Byte)
TL1 = 0x68; // Timer 1 initial value (Low Byte)

**Illustration of 50ms precise delay using the hardware timer**

## Example: Generating a precise 50 ms delay

A ‘flashing LED’ example based on a precise 50 ms hardware delay

```
/*------------------------------------------------------------------*-

   DELAY_HARDWARE_One_Second()

   Hardware delay of 1000 ms.

   *** Assumes 12MHz 8051 (12 osc cycles) ***

-*------------------------------------------------------------------*/
void DELAY_HARDWARE_One_Second(void)
   {
   unsigned char d;

   // Call DELAY_HARDWARE_50ms() twenty times
   for (d = 0; d < 20; d++)
      {
      DELAY_HARDWARE_50ms();
      }
   }


/*------------------------------------------------------------------*-

   DELAY_HARDWARE_50ms()

   Hardware delay of 50ms.
   *** Assumes 12MHz 8051 (12 osc cycles) ***

-*-----------------------------------------------------------*/
void DELAY_HARDWARE_50ms(void)
   {
   // Configure Timer 0 as a 16-bit timer
   TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
   TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)

   ET0 = 0; // No interupts

   // Values for 50 ms delay
   TH0 = 0x3C;  // Timer 0 initial value (High Byte)
   TL0 = 0xB0;  // Timer 0 initial value (Low Byte)

   TF0 = 0;          // Clear overflow flag
   TR0 = 1;          // Start timer 0

   while (TF0 == 0); // Loop until Timer 0 overflows (TF0 == 1)

   TR0 = 0;          // Stop Timer 0
   }
```

set up Timer 0, in Mode 1 (16-bit timer), without gating:

```
        TMOD &= 0xF0; // Clear all T0 bits (T1 left unchanged)
        TMOD |= 0x01; // Set required T0 bits (T1 left unchanged)
```

Note the use of the bitwise operators to change the state of SFR bits while leaving others unchanged: this is important as a different part of your program may be using Timer 1 for another purpose.

The overflow of a timer can be used to generate an interrupt: we have no need for this in the delay code presented here. We therefore disable interrupt generation as follows:

```
        ET0 = 0; // No interupts
```

Next, we load the timer registers with the initial timer value:

```
        TH0 = 0x3C; // Timer 0 initial value (High Byte)
        TL0 = 0xB0; // Timer 0 initial value (Low Byte)
```

In this case, we assume – again – the standard 12 MHz / 12 oscillations-per-instruction microcontroller environment. We require a 50 ms delay, so the timer requires the following number of increments before it overflows:

$$\frac{50ms}{1000ms} \times 1000000 = 50000 \text{ increments}$$

50*ms* ----------------~ 1000000 = 50000 increments
1000*ms*

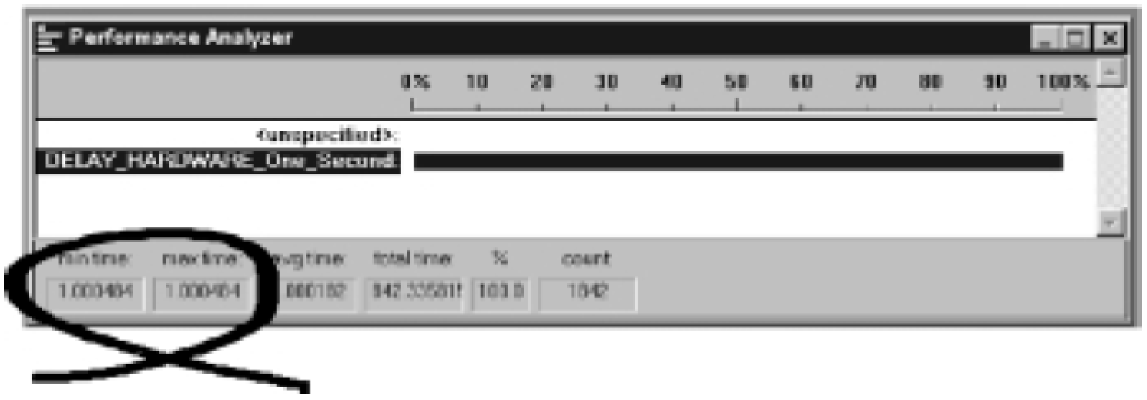
The timer overflows when it is incremented from its maximum count of 65535.
Thus, the initial value we need to load to produce a 50 ms delay is:

65536 – 50000 = 15536 (decimal) = 0x3CB0

Then we are ready to clear the timer flag, and start the timer running:

```
TF0 = 0; // Clear overflow flag
TR0 = 1; // Start timer 0
```

Using keil simulator to profile the code, In this case, executing the program in the hardware simulator confirms that the delays operate as required



```
Performance Analyzer
0%  10  20  30  40  50  60  70  80  90  100%
<unspecified>:
DELAY_HARDWARE_One_Second:
min time:  max time:  avg time:  total time:  %:     count:
1.000484   1.000484   [illegible]000182   842.335811   100.0   [illegible]
```

## Waveform Generation

- Using Logical Analyzer in Keil to view the waveform output

- Hands On:
    - Square
    - Triangle

- Other Waveforms:
    - Saw Tooth
    - Sine
    - Trapezoidal
    - Stair Case