What is a Timer

Timer is a clock that controls the sequence of an event while counting in fixed intervals of time. A Timer is used for producing precise time delay. Secondly, it can be used to repeat or initiate an action after/at a known period of time. This feature is very commonly used in several applications. An example could be setting up an alarm which triggers at a point of time or after a period of time.

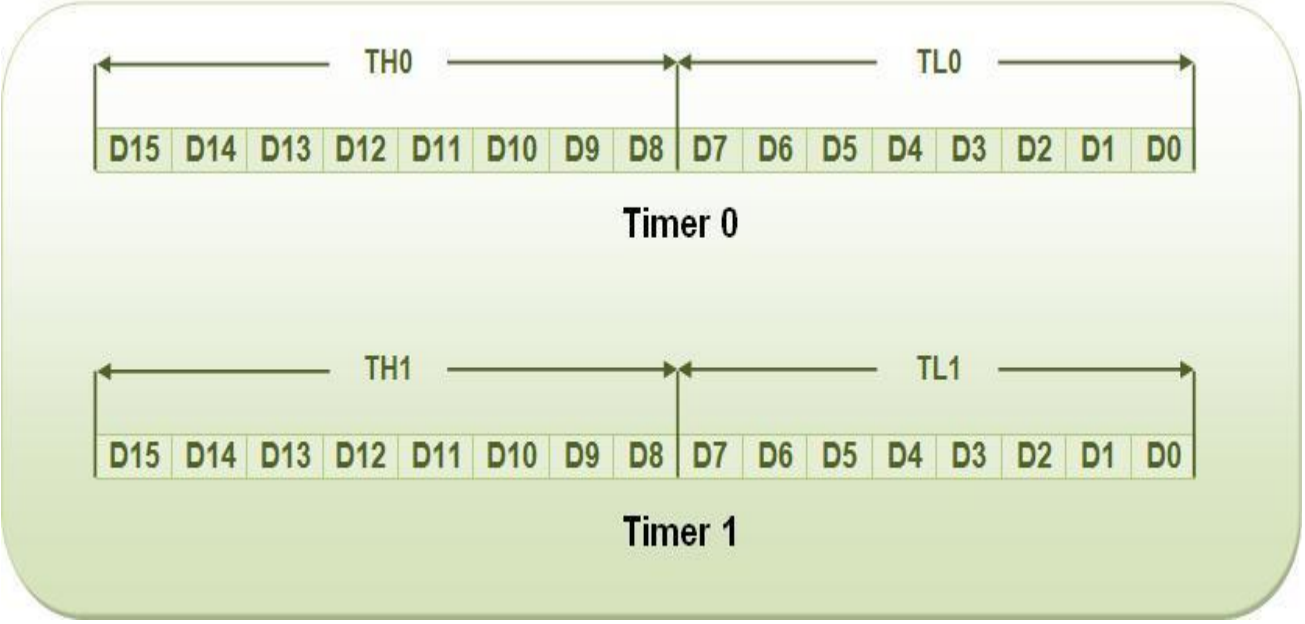
Timers in a controller: Why to use them

Most of the microcontrollers have inbuilt Timers. Timers in a controller not only generate time delays but they can also be used as counters. They are used to count an action or event. The value of counter increases by one, every time its corresponding action or event occurs. Timers in a controller are inbuilt chips that are controlled by **special function registers** (SFRs) assigned for Timer operations. These SFRs are used to configure Timers in different modes of operations.

While working with microcontrollers, it is more than often required to generate time delays. There are two possible ways of generating time delays. First is by using the code, like using *for* or *while* loops in a C program. However, the delays provided by the software **are not very precise**. The other method is to use Timers. Timers provide **time delays** that are very precise and accurate.

8051 Timers and registers

AT89C51 microcontroller has two Timers designated as Timer0 and Timer1. Each of these timers is assigned a 16-bit register. The value of a Timer register increases by one every time a timer counts. Timer takes a time period of one machine cycle to count one. (Machine cycle is a unit that refers to the time required by the microcontroller to execute instructions.) This means that the **maximum number of times** a timer can count without repeating is 2^{16}, i.e.,**65536**. So the maximum allowed counts in value of Timer registers can be from 0000H to FFFFH. Since **8051 is an 8 bit controller,** the registers of 8051 Timers are accessed as two different registers; one for lower byte and other for higher byte. For example, register of Timer0 is accessed as **TL0** for lower byte and **TH0** for higher byte. Similarly **TL1** and **TH1** are registers assigned to Timer 1.



8051 Timer Issues

While using 8051 Timers certain factors need to be considered, like whether the Timer is to be used for time keeping or for counting; whether the source for time generation is external clock or the controller itself; how many bits of Timer register are to be used or left unused.

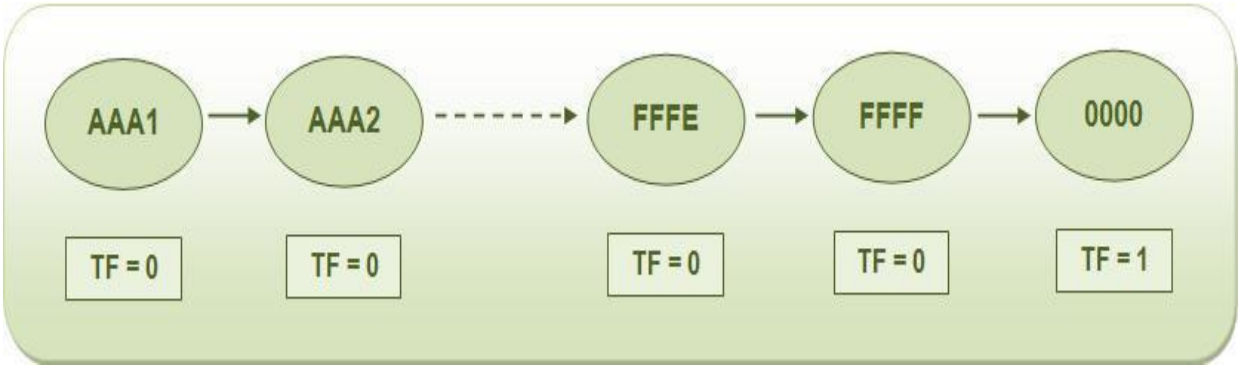
Timers & 8051 Timer Programming

How a Timer functions

The registers of Timers are loaded with some initial value. The value of a Timer register increases by one after every machine cycle. One machine cycle duration is the 1/12th of the frequency of the crystal attached to the controller.

For example, if the frequency of the crystal is 12 MHz, then the frequency for Timer will be 1MHz (1/12 of crystal frequency) and hence the time (T = 1/f) taken by the Timer to count by one is **1µs** (1/1MHz). Similarly if an 11.0592 MHz crystal is used, operating frequency of Timer is 921.6 KHz and the time period is **1.085 µs**.
If no value is loaded into the Timer, it starts counting from 0000H.

When the Timer reaches FFFFH, it reloads to 0000H. This **roll over** **is communicated to the controller by raising a flag** corresponding to that Timer, i.e., a flag bit is raised (set high) when the timer starts counting from 0000H again. **TF0** and **TF1** are the Timer flags corresponding to Timers 0 and 1. These flags must be cleared (set low) by software every time they are raised. The Timer may terminate updating register values after a roll over or continue with its operation.



Starting or stopping a Timer

For every Timer, there is a corresponding Timer control bit which can be set or cleared by the program to start or stop the Timer. **TR0** and **TR1** are the control bits for Timers 0 and 1 respectively. Setting the control bit would start the Timer.

TR0 = 1; starts Timer 0
TR1 = 1; starts Timer 1

Clearing the control bit would stop the Timer.

TR0 = 0; stops Timer 0
TR1 = 0; stops Timer1

Timer Programming

Configuring a Timer

A register called **TMOD** is used for configuring the Timers for the desired operation. TMOD is an 8-bit register with following bit configuration:

The lower four bits (TMOD.0 – TMOD.3) are used to configure Timer 0 while the higher four bits (TMOD.4 – TMOD.7) are for Timer 1. When GATE is high, the corresponding Timer is enabled only when there is an interrupt at corresponding INTx pin of AT89C51 controller and Timer control bit is high. Otherwise only setting Timer control bit is sufficient to start the Timer.

If C/T is low, Timer is used for time keeping, i.e., Timer updates its value automatically corresponding to 8051 clock source. When C/T is high, Timer is used as counter, i.e., it updates its value when it receives pulse from outside the 8051 controller.

M1 and M0 bits decide the Timer modes. There are four Timer modes designated as Modes 0, 1, 2 and 3.

Modes 1 and 2 are most commonly used while working with Timers.

TMOD = 0x01; sets the mode1 of Timer0 used for timing
TMOD = 0x20; sets the mode2 of Timer1 used for timing

Programming 8051 Timers

The programming of 8051 Timers can be done by using either polling method or by using interrupt. In polling, the microcontroller keeps monitoring the status of Timer flag. While doing so, it does no other operation and consumes all its processing time in checking the Timer flag until it is **raised on a rollover**. In interrupt method controller responds to only when the Timer flag is raised. The interrupt method prevents the wastage of controller's processing time unlike polling method.
Polling is mostly used for time delay generation and interrupt method is more useful when waveforms are to be generated or some action has to be repeated in fixed delays.

(i) Polling Method

The polling method involves the following algorithm:
1. Configure the Timer mode by passing a hex value into the TMOD register. This will tell the controller about which Timer is be used; the mode of Timer; operation (to be used as timer or counter); and whether external interrupt is required to start Timer.
2. Load the initial values in the Timer low TLx and high THx byte. (x = 0/1)
3. Start the Timer by setting TRx bit.
4. Wait while the Timer flag TFx is raised.
5. Clear the Timer flag. The Timer flag is raised when Timer **rolls over** from FFFFH to 0000H. If the Timer is not stopped, it will start updating from 0000H in case of modes 0 & 1 while with initial value in case of mode 2. If TFx is not cleared, controller will not be able to detect next rollover.
6. Stop the Timer by clearing TRx bit. If TRx bit is not cleared the Timer will restart updating from 0000H after the rollover in case of modes 0 and 1 while with initial value in case of mode 2.

(ii) Interrupt Method

The interrupt method makes use of a register called **Interrupt Enable** (IE) register. An 8051 microcontroller has 6 hardware interrupts. The interrupts refer to a notification, communicated to the controller, by a hardware device or software, on receipt of which controller skips temporarily whatsoever it was doing and responds to the interrupt.

The controller starts the execution of an **Interrupt Service Routine** (ISR) or Interrupt Handler which is a piece of code that tells the processor or controller **what to do on receipt of an interrupt**. After the execution of ISR, controller returns to whatever it was doing earlier (before the interrupt was received).

The Interrupt Enable register has the following bits which enable or disable the hardware interrupts of 8051 microcontroller.

IE :	EA	-	ET2	ES	ET1	EX1	ET0	EX0

When EA (IE.7) is set (=1), interrupts are enabled. When clear (EA=0), interrupts are disabled and controller does not respond to any interrupts.

ET0, ET1 & ET2 (IE.3, IE.4 & IE.5) are Timer interrupt bits. When set (high) the timer are enabled and when cleared (low) they are disabled. (8052 controllers have three Timers, so ET2 is its Timer 2

interrupt bit.) The ISR corresponding to these interrupts are executed when the TFx flags of respective Timers are raised. For more details on other IE register bits, refer Interrupts with 8051.

Note that IE register is bit addressable.

Timer programming using Timer interrupts involves following algorithm.

1. Configure the Timer mode by passing a hex value to TMOD register.
2. Load the initial values in the Timer low TLx and high THx byte.
3. Enable the Timer interrupt by passing hex value to IE register or setting required bits of IE register. For example,
   IE = 0x82; enables Timer 0 interrupt
   IE = 0x88; enables Timer 1 interrupt

   or
   EA = 1;
   ET0 = 1; enables Timer 0 interrupt

   IE^7 = 1;
   IE^3 = 1; enables Timer 1 interrupt
4. Start the Timer by setting TRx bit.
5. Write Interrupt Service Routine (ISR) for the Timer interrupt. For example,

ISR definition for Timer 0 :

```
void ISR_Timer0(void) interrupt 1
{
<Body of ISR>
}
```

ISR definition for Timer 1 :

```
void ISR_Timer1(void) interrupt 3
{
<Body of ISR>
}
```

6. If the Timer has to be stopped after once the interrupt has occurred, the ISR must contain the statement to stop the Timer.
For example,

```
void ISR_Timer1(void) interrupt 3
{
<Body of ISR>
TR1 =0;
}
```

7. If a routine written for Timer interrupt has to be repeated again and again, the Timer run bit need not be cleared. But it should be kept in mind that **Timer will start updating from 0000H** and **not the initial values in case of mode 0 and 1**. So the initial values must be reloaded in the interrupt service routine.
   For example,

```
void ISR_Timer1(void) interrupt 3
{
<Body of ISR>
TH1 =0XFF;           //load with initial values if in mode 0 or 1
TL1 = 0xFC;
}
```

Different modes of a Timer

There are four Timer modes designated as Modes 0, 1, 2 and 3. A particular mode is selected by configuring the M1 & M0 bits of TMOD register.

Mode	M1	M0	Operation
Mode 0	0	0	13-bit Timer
Mode 1	0	1	16-bit Timer
Mode 2	1	0	8-bit Auto Reload
Mode 3	1	1	Split Timer Mode

(i) Mode 0 : 13-bit Timer

Mode 0 is a 13 bit Timer mode and uses 8 bits of high byte and 5 bit prescaler of low byte. The value that the Timer can update in mode0 is from 0000H to 1FFFH. The 5 bits of lower byte append with the bits of higher byte. The Timer rolls over from 1FFFH to 0000H to raise the Timer flag.

(ii) Mode 1 : 16-bit Timer

Mode1 is one of the most commonly used Timer modes. It allows all 16 bits to be used for the Timer and so it allows values to vary from 0000H to FFFFH.

If a value, say YYXXH, is loaded into the Timer bytes, then the delay produced by the Timer will be equal to the product :
[(**FFFFH – YYXXH +1**) x (**period of one timer clock**)].

It can also be considered as follows: convert YYXXH into decimal, say NNNNN, then delay will be equal to the product :
[(**65536-NNNNN**) x (**period of one timer clock**)].
The period of one timer clock is 1.085 µs for a crystal of 11.0592 MHz frequency as discussed above.

Now **to produce a desired delay**, divide the required delay by the Timer clock period. Assume that the division yields a number NNNNN. This is the number of times Timer must be updated before it

stops. Subtract this number from 65536 (binary equivalent of FFFFH) and convert the difference into hex. This will be the initial value to be loaded into the Timer to get the desired delay.
The calculator application in Windows can be a handy tool to carry out these calculations.

Example code
Time delay in Mode1 using polling method

```
// Use of Timer mode 1 for blinking LED using polling method
//              XTAL              frequency              11.0592MHz
#include<reg51.h>
sbit led = P1^0;                          //  LED  connected  to  1st
pin                  of                    port                   P1
void                                                          delay();

main()
{
       unsigned                          int                          i;
       while(1)
       {
       led=~led;                          //        Toggle        LED
       for(i=0;i<1000;i++)
       delay();                           //        Call        delay
       }
}

void delay()                              // Delay generation using
Timer                  0                    mode                   1
{
       TMOD = 0x01;                       //    Mode1    of    Timer0
       TH0= 0xFC;                         //   FC66   evaluated   hex
value              for              1millisecond               delay
       TL0                              =                          0x66;
       TR0 = 1;                           //       Start       Timer
       while(TF0 == 0);                   //  Using  polling  method
       TR0 = 0;                           //        Stop        Timer
       TF0 = 0;                           //        Clear        flag
}
```

Example code
Time delay in Mode1 using interrupt method

```
// Use of Timer mode 1 for blinking LED with interrupt method
// XTAL frequency 11.0592MHz
#include<reg51.h>
sbit LED = P1^0;                          // LED connected to 1st
pin of port P1
void Timer(void) interrupt 1                    // Interrupt No.1
for Timer 0
{
```

```
        led=~led;                               // Toggle LED on interrupt
}

main()
{
        TMOD = 0x01;                            // Mode1 of Timer0
        TH0=0x00;                               // Initial values loaded
to Timer
        TL0=0x00;
        IE = 0x82;                              // Enable interrupt
        TR0=1;                                  // Start Timer
        while(1);                               // Do nothing
}
```

(iii) Mode 2 : 8-bit Auto Reload

Mode 2 is an 8 bit mode. The initial value is loaded into the higher byte. A copy of the same is passed to the lower byte. The Timer can update from 00H to FFH. The Timer rolls over from FFH to initial value automatically.

Mode 2 is commonly used for setting baud rates for serial communication.

Example code

Time delay in Mode2 using polling method

```
// Use of Timer mode 2 for blinking LED with polling method
// XTAL frequency 11.0592MHz
#include<reg51.h>
sbit led = P1^0;                                // LED connected to 1st
pin of port P1void delay();
main()
{
        unsigned int i;
        while(1)
        {
        led=~led;                               // Toggle LED
        for(i=0;i<1000;i++)
        delay();                                // Call delay
        }
}

void delay()
{
        TMOD = 0x02;                            // Mode1 of Timer0
        TH0= 0xA2;                              // Initial value loaded to
Timer
        TR0 = 1;                                // Start Timer
        while(TF0 == 0);                        // Polling for flag bit
        TR0 = 0;                                // Stop Timer
        TF0 = 0;                                // Clear flag
}
```

(iv) Mode 3 : Split Timer

In mode 3, also known as split mode, the Timer breaks into two 8-bit Timers that can count from 00H up to FFH. The initial values are loaded into the higher byte and lower byte of the Timer. In this case the start and flag bits associated with the other Timer are now associated with high byte Timer. So one cannot start or stop the other Timer. The other Timer can be used in modes 0, 1 or 2 and is updated automatically for every machine cycle.

For example, if Timer0 is used in split mode, TH0 and TL0 will become separate Timers. The start and flag bits associated with Timer1 will now be associated with the TH0. Timer 1 cannot be stopped or started, but will be updated for every machine cycle.

Split mode is useful when two Timers are required in addition to a baud rate generator.