Data types and Time Delay in 8051 C

Embedded C is one of the most popular and most commonly used Programming Languages in the development of Embedded Systems. It is popular due to its efficiency, less development time and portability.

Data Types in Embedded C

Data Types in C Programming Language help us declaring variables in the program. The data types in Embedded C are-

a) unsigned char
b) signed char
c) signed int
d) unsigned int
e) Sbit(Single bit)
f) Bit
g) Sfr

Unsigned Char:

Unsigned char is 8 bit data type in the range 0 – 255 (0 - FF). This is widely used because 8051 is a 8 bit microcontroller. This is used as ASCII characters and to count the value

Signed char:

Signed char is an 8 bit data type in which the MSB represents the sign (+ or -) in the range -128 to +127.

Unsigned int:

Unsigned int is a 16 bit data type which takes the value in the range 0 to 65535 (0 – FFFF). This is used for –

- Defining 16 bit memory addresses
- To set the counter values which is more than 256

Signed int:

Signed int is a 16 bit data type. MSB is the sign bit and the remaining 15 bits represent the magnitude from - 32768 to +32767.

sbit:

This data type is used in case of accessing a single bit addressable registers. It allows to access to the single bit SFR registers .

Bit:

This data type is used for accessing the bit addressable memory of RAM (20h-2fh).

sfr:

This data type is used for accessing a SFR register by another name. All the SFR registers must be declared with capital letters

Program 1:

Write an 8051 C program to send hex values for ASCII characters of 0, 1, 2, 3, 4, 5, A, B. C, and D to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mynum[]= "012345ABCD";
    unsigned char z;
    for(z=0;z<=10;z++)
      P1=mynum[z];
```

Program 2:

Write an 8051 C program to send values 00 - FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char z;
    for(z=0;z<=255;z++)
      P1=z;
  }
```

Program 3:

Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
//sign numbers
#include <reg51.h>
void main(void)
  {
    char mynum[]= {+1,-1,+2,-2,+3,-3,+4,-4};
    unsigned char z;
    for(z=0;z<=8;z++)
      P1=mynum [z];
  }
```

Program 4:

Write an 8051 C program to toggle all bits of P1 continuously

```
// toggle P1 forever
#include <reg51.h>
Void main(void)
  {
    for(;;)        //repeat forever
     {
       P1=0x55;   //0x indicates the data is in hex (binary)
       P1=0xAA;
     }
  }
```

Program 5:

Write an 8051 C program to toggle bit D0 of the port 1 50000 times

Solution:

```
#include <reg51.h>
sbit MYBIT = P1^0;     //notice that sbit is
                       //declared outside of main
void main(void)
  {
    unsigned int z;
    for (z=0; z<=50000; z++)
      {
        MYBIT = 0;
        MYBIT = 1;
      }
  }
```

Widely used data types in 8051 C

Data Type	Size in Bits	Data Range/Usage
unsiged char	8-bit	0 to 255
(signed) char	8-bit	–128 to +127
unsigned int	16-bit	0 to 65535
(signd) int	16-bit	–32,768 to +32,767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 - FFH only

Time delay

There are two methods to create a time delay 8051 C

a) Using a simple for loop
b) Using the 8051 timers

Time delay using for loop:

In creating a time delay using a for loop, there are three factors that can affect the accuracy of the delay.

1. The 8051 design.
2. The crystal frequency
3. Compiler choice.

1. **The 8051 design:**

The number of machine cycles and the number of clock periods per machine cycle vary among different versions of the 8051/52 microcontroller. While the original 8051/52 design used 12 clock periods per machine cycle, many of the newer generations of the 8051 use fewer clocks per machine cycle.

2. **The crystal frequency**

The crystal frequency connected to the XI – X2 input pins. The duration of the clock period for the machine cycle is a function of this crystal frequency.

3. **Compiler choice.**

If the program is in Assembly language, then the exact instructions and their sequences used in the delay subroutine can be controlled. In the case of C programs, it is the C compiler that converts the C statements and functions to Assembly language instructions. As a result, different compilers produce different code. In other words, if we compile a given 8051 C programs with different compilers, each compiler produce different hex code.

1. **Write an 8051 C program to toggle bits of PI continuously forever with some delay.**

Solution:

```
// Toggle PI forever with some delay in between “on” and “off”,

#include <reg51.h>
```

```
void main(void)
  {
    unsigned int x;
    for(;;)                        //repeat forever
      {
        P1=0x55;
        for(x=0;x<40000;x++);   //delay size unknown
        P1=0xAA;
        for(x=0;x<40000;x++);
    }
  }
```

2. **Write an 8051 C program to toggle the bits of PI ports continuously with a 250 ms delay.**

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    while(1)    //repeat forever
      {
        P1=0x55;
        MSDelay(250);
        P1=0xAA;
        MSDelay(250);
      }
  }

void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<1275;j++);
  }
```

Program 3:

Write a 8051 C program to toggle all the bits of P0 and P2 continuously with a 250 ms delay.

Solution:

```
//This program is tested for the DS89C420 with XTAL = 11.0592 MHz
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    while(1)    //another way to do it forever
      {
        P0=0x55;
        P2=0x55;
        MSDelay(250);
        P0=0xAA;
        P2=0xAA;
        MSDelay(250);
      }
  }
void MSDelay(unsigned int itime)
  {
    unsigned int i, j;
    for(i=0;i<itime;i++)
      for(j=0;j<1275;j++);
  }
```

Data Conversion Programs in 8051 C

Application of logic and rotate instructions in the conversion of BCD and ASCII.

ASCII numbers

On ASCII keyboards, when the key "0″ is activated, "011 0000″ (30H) is provided to the computer. Similarly, 31H (011 0001) is provided for the key "1″, and so on, as shown in Table

ASCII Code for Digits 0 – 9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

Packed BCD to ASCII conversion

The RTC (Real Time Clock) provides the time of day (hour, minute, second) and the date (year, month, day) continuously, regardless of whether the power is on or off. However, this data is provided in packed

BCD. To convert packed BCD to ASCII, it must first be converted to unpacked BCD. Then the unpacked BCD is tagged with 011 0000 (30H). The following demonstrates converting from packed BCD to ASCII.

Packed BCD	Unpacked BCD	ASCII
0x29	0x02, 0x09	0x32, 0x39
00101001	00000010,00001001	00110010,00111001

ASCII to packed BCD conversion

To convert ASCII to packed BCD, it is first converted to unpacked BCD (to get rid of the 3), and then combined to make packed BCD.

For example, 4 and 7 on the keyboard give 34H and 37H, respectively. The goal is to produce 47H or "0100 0111″, which is packed BCD.

Key	ASCII	Unpacked BCD	Packed BCD
4	34	00000100	
7	37	00000111	01000111 or 47H

After this conversion, the packed BCD numbers are processed and the result will be in packed BCD format

Program 1:

Write an 8051 C program to convert ASCII digits of '4' and '7' to packed BCD and display them on P1.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char bcdbyte;
    unsigned char w='4';
    unsigned char z='7';
    w = w & 0x0F;     //mask 3
    w = w << 4;       //shift left to make upper BCD digit
    z = z & 0x0F;     //mask 3
    bcdbyte = w | z; //combine to make packed BCD
    P1 = bcdbyte;
  }
```

Program 2:

Write an 8051 C program to convert packed BCD 0x29 to ASCII and display the bytes on P1 and P2.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char x, y, z;
    unsigned char mybyte = 0x29;
    x = mybyte & 0x0F;          //mask lower 4 bits
    P1 =  x | 0x30;             //make it ASCII
    y = mybyte & 0xF0;          //mask upper 4 bits
    y = y >> 4;                 //shift it to lower 4 bits
    P2 = y | 0x30;              //make it ASCII
  }
```

DAC Interfacing

Digital-to-Analog (DAC) converter

The digital-to-analog converter (DAC) is a device used to convert digital pulses to analog signals. In this section we discuss the basics of interfacing a DAC to the 8051.
There are two methods of creating a DAC:

1. binary weighted method and
2. R/2R ladder method

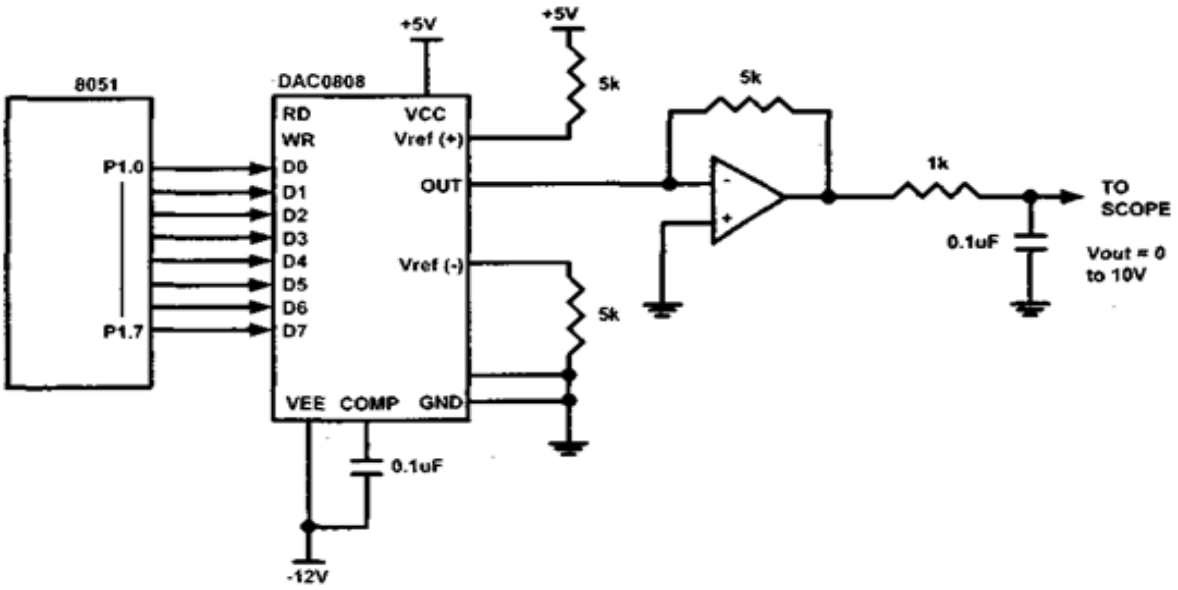
The majority of integrated circuit DACs use the R/2R method since it can achieve a much higher degree of precision. The first criterion for judging a DAC is its **resolution, which is a function of the number of binary inputs.** The number of data bit inputs decides the resolution of the DAC. An 8-input DAC provides 256 discrete voltage (or current) levels of output.

MC1408 DAC (or DAC0808)

In the MC1408 (DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the I_{out} pin, we convert the result to voltage.
The total current provided by the I_{out} pin is a function of the binary numbers at the D0 – D7 inputs of the DAC0808 and the reference current (I_{ref}), which is given by :

$$I_{out} = I_{ref}\left(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256}\right)$$

Where D0 is the LSB, D7 is the MSB for the inputs, and I_{ref} is the input current that must be applied to pin 14. The I_{ref} current is generally set to 2.0 mA. The following figure shows the generation of current reference (setting $I_{ref} = 2$ mA) by using the standard 5-V power supply and standard resistors.



Problem-1:

Assuming that R = 5K and I_{ref} = 2 mA, calculate V_{out} for the following binary inputs:
(a) 10011001 binary (99H) (b) 11001000 (C8H)

Solution:

(a) I_{out} = 2 mA (153/256) = 1.195 mA and V_{out} = 1.195 mA × 5K = 5.975 V
(b) I_{out} = 2 mA (200/256) = 1.562 mA and V_{out} = 1.562 mA × 5K = 7.8125 V

Converting I_{out} to voltage in DAC0808

Ideally we connect the output pin I_{out} to a resistor, convert this current to voltage. This can cause inaccuracy since the input resistance of the load where it is connected will also affect the output voltage. For this reason, the I_{ref} current output is isolated by connecting it to an op-amp 741 with R_f = 5K ohms for the feedback resistor.

Program:

Write a program to send data to the DAC to generate a stair-step ramp.

Solution: Refer Manual

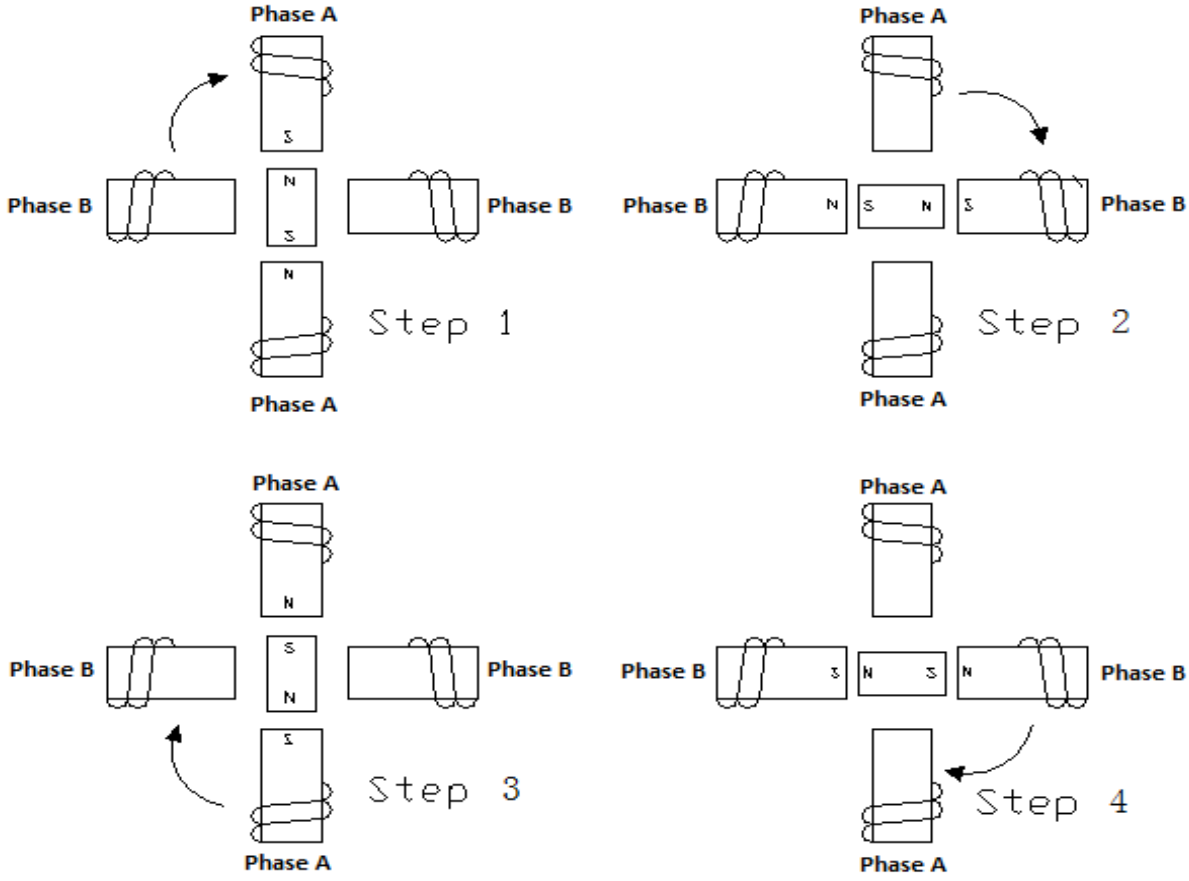
STEPPER MOTOR INTERFACING

Stepper motors

A *stepper motor* translates electrical pulses into mechanical movement. It finds applications in disk drives, dot matrix printers, and robotics, for position control.

Construction:

Stepper motors have a permanent magnet ***rotor*** (also called the *shaft)* surrounded by a ***stator*.** They have four stator windings that are paired with a centre-tapped common as shown in Figure 2. This type of stepper motor is commonly referred to as *a. four-phase* or unipolar stepper motor. The centre tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator. Notice that while a conventional motor shaft runs freely, the stepper motor shaft moves in a fixed repeatable increment, which allows one to move it to a precise position. The stepper motor shaft moves in a fixed repeatable increment which is based on the principle that poles of the same polarity repel and opposite poles attract. The direction of rotation is decided by the stator poles and the stator poles are determined by current sent through the wire coils. As the direction of current is changed, the polarity is also changed causing the reverse motion of the rotor. The 2-phase, 4-step stepping sequence is as shown below.



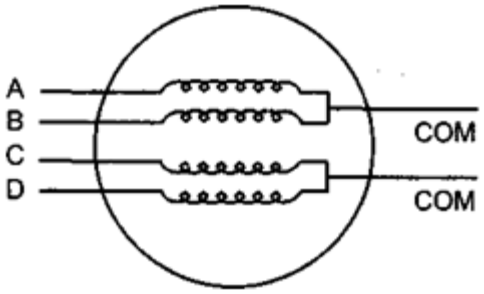
The following table shows a 2-phase, 4-step stepping sequence.

Normal 4-Step Sequence

Clockwise

Step #	Winding A	Winding B	Winding C	Winding D
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

Counter-clockwise



This is called 4-step switching sequence since after four steps the same two windings will be 'ON'. After completing every four steps, the rotor moves only one tooth pitch. Therefore, in a stepper motor with 200 steps per revolution the rotor has 50 teeth since 4 x 50 = 200 steps are needed to complete one revolution. Hence the minimum step angle is the function of number of teeth on the rotor. I.e. smaller the step angle, the more teeth the rotor passes.

Step angle

The step angle is the minimum degree of rotation associated with a single step. Various motors have different step angles.
Steps per revolution is the total number of steps needed to rotate one complete rotation or 360 degrees.
The following table shows the various step angles and steps per revolution.

Table 17-4: Stepper Motor Step Angles

Step Angle	Steps per Revolution
0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

Relation between steps per second and rpm:

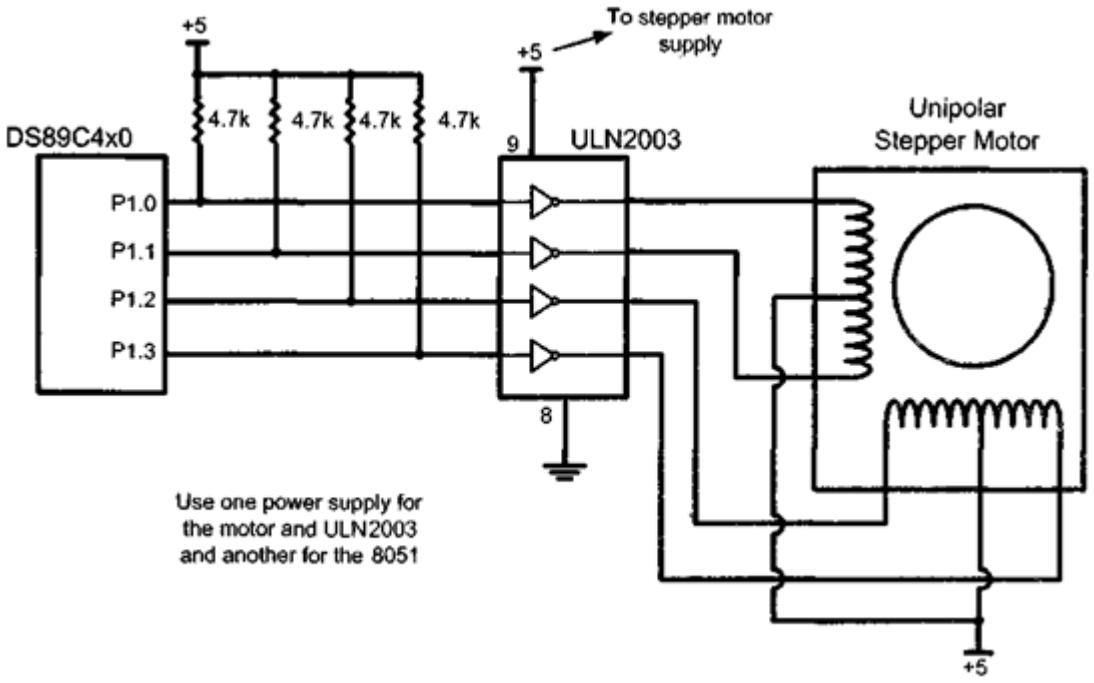
The relation between rpm (revolutions per minute), steps per revolution, and steps per second is as follows;

$$\text{Steps per second} = \frac{rpm\, x\, steps\ per\ revolutioin}{60}$$

Motor speed:
The motor speed, measured in steps per second, is a function of the switching rate.

Holding torque:
Holding torque is defined as follows." With the motor shaft at standstill or zero rpm condition, the amount of torque from the external source required to break away the shaft from its holding position". It is measured with rated voltage and current applied to the motor. The unit is ounce-inch or kg-cm

8051 Connection to Stepper Motor



Steps per second and rpm relation

The relation between rpm (revolutions per minute), steps per revolution, and steps per second is as follows.

$$Steps\ per\ second = \frac{rpm \times Steps\ per\ revolution}{60}$$

The four-step sequence and number of teeth on rotor

The switching sequence shown earlier in Table 17-3 is called the 4-step switching sequence since after four steps the same two windings will be "ON" How much movement is associated with these four steps? After completing every four steps, the rotor moves only one tooth pitch. Therefore, in a stepper motor with 200 steps per revolution, the rotor has 50 teeth since 4×50 = 200 steps are needed to complete one revolution. This leads to the conclusion that the minimum step angle is always a function of the number of teeth on the rotor. In other words, the smaller the step angle, the more teeth the rotor passes. See Example 17-2.

Motor speed

The motor speed, measured in steps per second (steps/s), is a function of the switching rate. Notice in Example 17-1 that by changing the length of the time delay loop, we can achieve various rotation speeds.

Holding torque

The following is a definition of holding torque: “With the motor shaft at standstill or zero rpm condition, the amount of torque, from an external source, required to break away the shaft from its holding position. This is measured with rated voltage and current applied to the motor.” The unit of torque is ounce-inch (or kg-cm).

JUMP , LOOP AND CALL INSTRUCTIONS

The jumps and calls are Decision codes that alter the flow of program by examining the results of the action codes and altering the contents of the program counter.

A jump permanently changes the contents of the program counter with a new address number if certain program conditions exist. The difference in bytes of the new address from the address in the program where the jump (or Call) is located is called "Range" of jump or call.

LOOP instruction:

Repeating a sequence of instructions a certain number of times is called a loop. An instruction DJNZ reg, label is used to perform a Loop operation. In this instruction, a register is decremented by 1; if it is not zero, then 8051 jumps to the target address referred to by the label.

Eg: **Program 1**

1. Write an ALP to (a) to clear the Accumulator

   (b) Add 3 to Accumulator ten times.

```
        MOV A, #00
        MOV R2, #10
Again:  ADD A,  #03
        DJNZ R, Again
        MOV R5, A
```

Program 2.

Write a program to clear 16 RAM locations starting at RAM address 60H.

Solution:

```
        CLR     A           ;A=0
        MOV     R1,#60H     ;load pointer. R1=60H
        MOV     R7,#16      ;load counter, R7=16 (10 in hex)
AGAIN:  MOV     @R1,A       ;clear RAM location R1 points to
        INC     R1          ;increment R1 pointer
        DJNZ    R7,AGAIN    ;loop until counter = zero
```

Program 3:

Write a program to copy the value 55H into RAM memory locations 40H to 45H using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) with a loop.

Solution:

(a)

```
MOV A,#55H      ;load A with value 55H
MOV 40H,A       ;copy A to RAM location 40H
MOV 41H,A       ;copy A to RAM location 41H
MOV 42H,A       ;copy A to RAM location 42H
MOV 43H,A       ;copy A to RAM location 43H
MOV 44H,A       ;copy A to RAM location 44H
```

(b)

```
MOV A,#55H      ;load A with value 55H
MOV R0,#40H     ;load the pointer. R0=40H
MOV @R0,A       ;copy A to RAM location R0 points to
INC R0          ;increment pointer. Now R0=41H
MOV @R0,A       ;copy A to RAM location  R0 points to
INC R0          ;increment pointer. Now R0=42H
MOV @R0,A       ;copy A to RAM location  R0 points to
INC R0          ;increment pointer. Now R0=43H
MOV @R0,A       ;copy A to RAM location R0 points to
INC R0          ;increment pointer. Now R0=44H
MOV @R0,A
```

(c)

```
        MOV   A,#55    ;A=55H
        MOV   R0,#40H ;load pointer. R0=40H, RAM address
        MOV   R2,#05   ;load counter, R2=5
AGAIN:  MOV   @R0,A    ;copy 55H to RAM location R0 points to
        INC   R0       ;increment R0 pointer
        DJNZ  R2,AGAIN      ;loop until counter = zero
```

Program 4:

Write a program to copy a block of 10 bytes of data from RAM locations starting at 35H to RAM locations starting at 60H.

Solution:

```
        MOV   R0,#35H    ;source pointer
        MOV   R1,#60H    ;destination pointer
        MOV   R3,#10     ;counter
BACK:   MOV   A,@R0      ;get a byte from source
        MOV   @R1,A      ;copy it to destination
        INC   R0         ;increment source pointer
        INC   R1         ;increment destination pointer
        DJNZ  R3,BACK    ;keep doing it for all ten bytes
```

Nested loop:

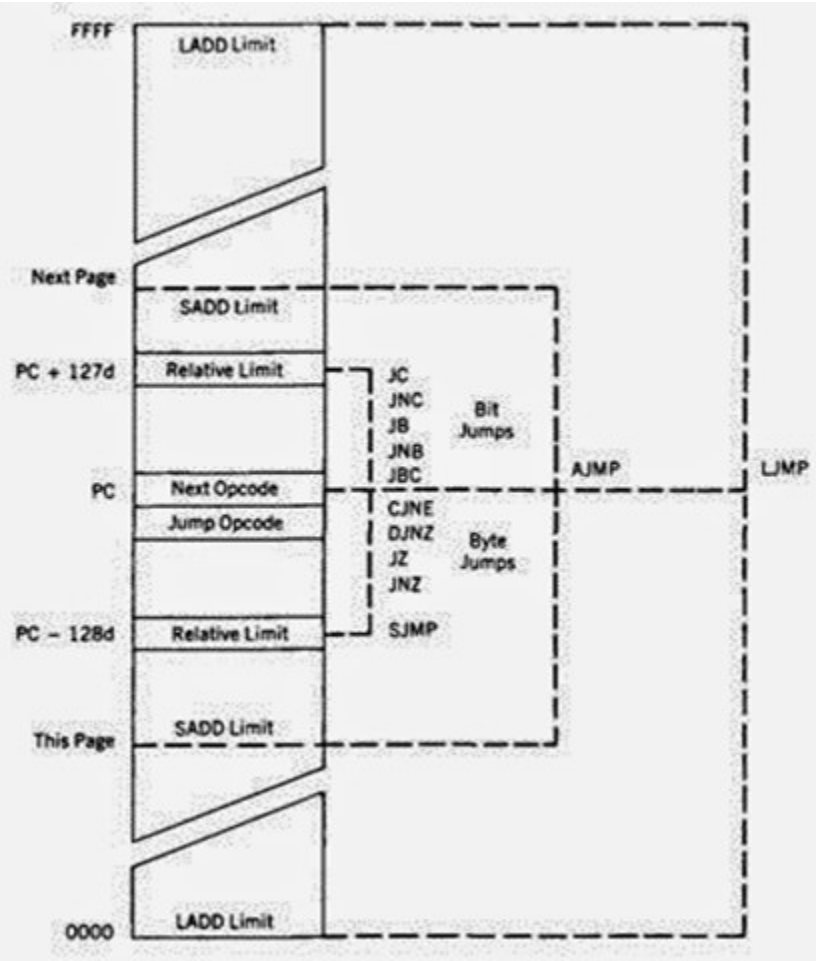
```
        MOV A, #55H ;  A= 55 hex
        MOV R1, #100 ; the outer counter R1 =100
NEXT:   MOV R2, # 20 ; the inner counter AGAIN:
        CPL A, # 05 ; add five to register  A
        DJNZ R2, AGAIN ;  repeat until R1=0 (100 times)
        DJNZ R1, NEXT ; repeat till 20 times (outer loop)
```

The jump and call instructions are referred to as branching instructions.

Jump or call instructions have 3 ranges;

- A RELATIVE RANGE - +127d to -128d bytes from the instruction following the jump or call instruction.
- ABSOLUTE RANGE - it is the range of address on the same 2K byte page as the instruction following the jump or call.
- LONG RANGE – it is the range of any addresses from 0000h to FFFFh anywhere in the program memory.

The following fig shows the relative range of all jump instructions:



Relative range:

In relative jumps the contents of the program counter is replaced with new address which lies +127d to -128d, relatively from the address of the instruction following the jump. The address placed in the PC is always relative to the address where the jump occurs. When we change the absolute address of the jump instruction, then jump address also changes, but remains at the same distance from the jump instruction.

Advantages of relative address jump:

1. Only one byte of data is needed to specify the relative address. The data is a positive integer or in 2`s complement to specify negative integers.
2. Specifying one byte saves program bytes and speeds up program execution.
3. When the program code is relocated in a different ROM location, the relative address does not change.

The only disadvantage is that all addresses jumped are in the range of +127d to -128d. If longer jumps are required, then a relative jump can be done to another relative jump until the required address is obtained.

Short absolute range:

In short absolute range, the program memory which is from 0000h to FFFFh is divided into logical divisions called "Pages" of convenient size. In 8051, the program memory is arranged as 2K pages, giving a total of 32d (20h) pages. The hexadecimal address of each page is as shown below.

Page	Address	Page	Address	Page	Address
00	0000 – 07FF	0B	5800 – 5FFF	16	B000 – B7FF
01	0800 – 0FFF	0C	6000 – 67FF	17	B800 – BFFF
02	1000 – 17FF	0D	6800 – 6FFF	18	C000 – C7FF
03	1800 – 1FFF	0E	7000 – 77FF	19	C800 – CFFF
04	2000 – 27FF	0F	7800 – 7FFF	1A	D000 – D7FF
05	2800 – 2FFF	10	8000 – 87FF	1B	D800 – DFFF
06	3000 – 37FF	11	8800 – 8FFF	1C	E000 – E7FF
07	3800 – 3FFF	12	9000 – 97FF	1D	E800 – EFFF
08	4000 – 47FF	13	9800 – 9FFF	1E	F000 – F7FF
09	4800 – 4FFF	14	A000 – A7FF	1F	F800 – FFFF
0A	5000 – 57FF	15	A800 – AFFF		

The upper 5-bits of the PC hold the page number and the lower 121 bits holds the address within the page. An Absolute address is formed by taking the page number of the instruction following the branch and attaching the absolute page range address of 11 bits to it form the 16 bit address.

Long Absolute range:

Addressing that can access the entire program space from 0000h to FFFFh use Long range addressing. Long range addresses require more bytes of code to specify and are relocatable only at the beginning of 64K pages.

Long range addressing has the advantage of using the entire program address space available to 8051.

JUMPS

Jumps operate by testing for conditions that are specified in the jump mnemonic. If the conditions specified are *true*, then jump occurs; if it is *false* then the instruction following the jump is executed.

There are two types of jumps:

- Unconditional jumps
- Conditional jumps.

Unconditional jump:

MNEMONIC	OPERATION
JMP @A+DPTR	Jump to address A + DPTR. The address can be anywhere in program memory.
AJMP *sadd*	Jump to absolute short address sadd. The instruction is 2 bytes long.
LJMP *ladd*	Jump to absolute long address ladd. The destination address is a 16 bit address. The instruction is 3 byte long. The destination can be anywhere in 64K program memory space.
SJMP *radd*	Jump to relative address radd. The jump distance is limited to a range of -128d to +127d relative to the instr5uction following the jump.
NOP	Do nothing and go to next instruction; NOP (no operation) is used to waste time in software timing loop.

Unconditional jump do not test for any bit or byte to determine whether the jump should be taken. The jump is always taken. Any range of jump can be found in this group. These are the only jumps that can jump to any location in memory.

Conditional jumps:

Conditional Jumps operate by testing for conditions that are specified in the jump mnemonic. If the conditions specified are *true*, then jump occurs; if it is *false* then the instruction following the jump is executed. There are 2 types of conditional jumps;

- Bit jumps
- Byte jumps.

Bit jumps:

All bit jumps operate according to the status of the Carry flag in the PSW or the status of any bit addressable location. The instructions are as follows

MNEMONIC	OPERATION
JC *radd*	Jump to relative address *radd if carry flag is set to 1*
JNC *radd*	Jump to relative address *radd if carry flag is reset to 0*
JB b, *radd*	Jump to relative address *radd if addressable bit b is set to 1*
JNB b, *radd*	Jump to relative address *radd if addressable bit b is reset to 0*
JBC b, *radd*	Jump to relative address *radd if addressable bit b is set to 1. Then clear the bit 0*

Note:

- In all of the above, the jump address location is contents of PC + relative address *addr.*
- The bit b must be bit addressable.

Byte jumps:

The byte jump instructions test a byte of data. If condition tested is true, the jump is executed. If the condition is false, the instruction after the jump is executed. All byte jumps use relative range of addressing. The instructions are as follows

MNEMONIC	OPERATION
CJNE A, add, radd	Compare the contents of the A register with the contents of the direct address; if they are NOT EQUAL, then jump to relative address; carry flag is set if A<contents of direct address; otherwise set the carry flag to 0
CJNE A, #n, radd	Compare the contents of the A register with the immediate number n; if they are NOT EQUAL, then jump to relative address; carry flag is set if A < the number ; otherwise set the carry flag to 0
CJNE Rn, #n, radd	Compare the contents of the register Rn with the immediate number n; if they are NOT EQUAL, then jump to relative address; carry flag is set if Rn < the number ; otherwise set the carry flag to 0
CJNE @Rp, #n, radd	Compare the contents of the address contained in register Rp with the immediate number n; if they are NOT EQUAL, then jump to relative address; carry flag is set if the contents of the address in Rp are < the number; otherwise set the carry flag to 0
DJNZ Rn, radd	Decrement register Rn by 1 and jump top relative address if the result is not 0;
DJNZ add, radd	Decrement the direct register Rn by 1 and jump top relative address if the result is not 0;
JZ radd	Jump to relative address if A is 0; the flags and the A register are not changed.
JNZ radd	Jump to relative address if A is not 0; the flags and the A register are not changed.

In general we can write the compare instructions as

CJNE destination, Source, relative address

Summary of conditional jump instructions:

Instruction	Action
JZ (Jump Zero)	Jump if A=0
JNZ (Jump no zero)	Jump if A≠0
DJNZ Rn,target	Decrement and jump if byte≠0
CJNE A,byte	Compare A with byte and jump if not equal (A≠byte)
CJNE reg,#data	Compare reg. with #data and jump if not equal (byte ≠ #data)
JC (Jump carry)	Jump if CY=1
JNC (Jump no carry)	Jump if CY=0
JB (Jump bit)	Jump if bit=1
JNB (Jump no bit)	Jump if bit=0
JBC (jump bi clear bit)	Jump if bit=1 and clear bit

CALLS

CALL instruction is used to alter the sequence of instruction. A program that does not deal with the outside world of the microcontroller could be written using jumps to alter the sequence of program execution. This method id called "POLLING"

Another method to alter the sequence of program execution is by using interrupts. When an interrupt occurs, the execution branches out to another location to execute a smaller program called the *subroutine*. When the interrupt has been serviced, execution resumes from where it had branched off.

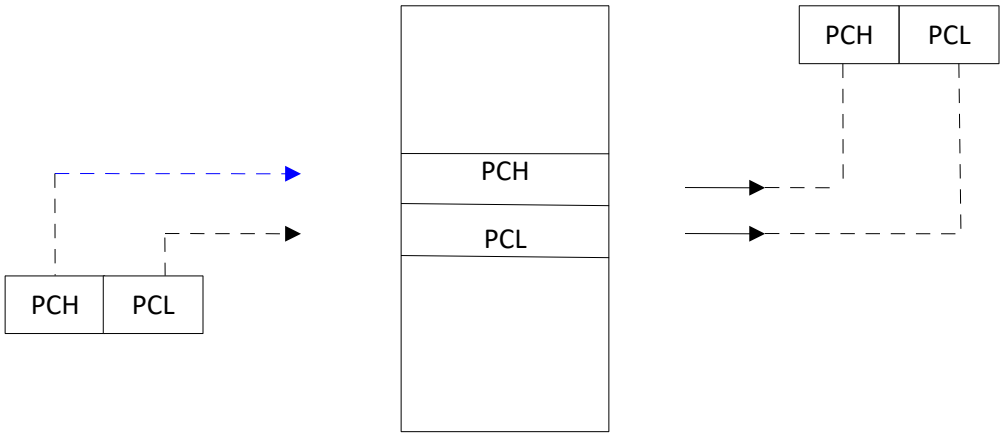
CALL instructions can be explicitly included in the program or implicitly included in interrupts. In either case the sequence of events which follow are the same.

Calls and the stack:

A CALL instruction generated either by hardware or software causes a jump to the address where the subroutine called is located. At the end of the subroutine the program resumes the operation at the opcodes address immediately following the CALL, after the subroutine has been executed. Therefore it is necessary to keep track of the address where the main program has branched off to execute the subroutine.

The STACK area of the internal RAM is used to store automatically the address called the "return address" of the instruction after the call. The stack pointer register holds the address of the last space used on the stack. It stores the return address above this space, adjusting itself upward as the return address is stored.

The following sequence of instructions is executed.



1. A CALL opcodes occurs in the program software, or an interrupt is generated in the hardware circuitry.
2. The return address of the next instruction after the CALL instruction or interrupt
3. The return address bytes are pushed on the stack, low byte first.
4. The stack pointer is incremented for each push on the stack
5. The subroutine address is placed in the program counter.
6. The subroutine is executed.
7. A RET (return) opcodes is encountered at the end of the subroutine.
8. Two pop operations restore the return address to the PC from the stack area in internal RAM.
9. The stack pointer is decremented for each address byte pop.

The 8051 has two instructions for CALL

LCALL:

The mnemonic “LCALL addr” is used for LCALL> this is a 3 byte instruction. The first byte us the op-code; the second and the third bytes are used for the address of the target subroutine. Hence, LCALL is used to call subroutines located anywhere within the 64K byte address space of the 8051.

ACALL:

The mnemonic is “ACALL addr”. ACALL is a 2 byte instruction. The target address must be within 2K bytes

Note:

Soft ware calls may use short-and-long range addressing. RET instruction is used at the end of subroutines called by LCALL or ACALL.

Interrupts are calls forced by hardware action. Call subroutines located at the predefined addresses in program memory. RET1 instruction is used to return from subroutine called by a hardware interrupt and reset the interrupt logic.

Timer Programming

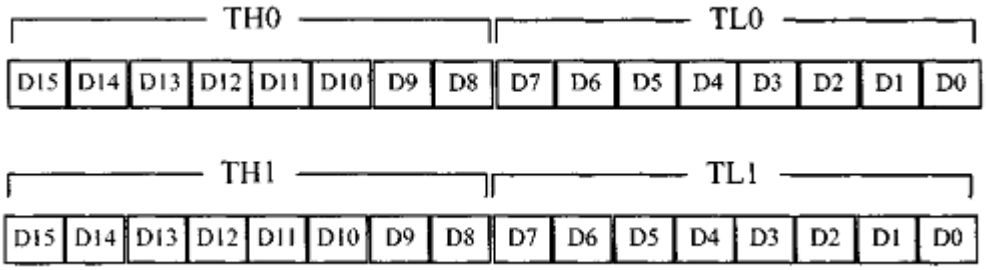
The 8051 has two timers: Timer 0 and Timer 1. They can be used either as timers or as event counters.

Basic registers of the timer

Both Timer 0 and Timer 1 are 16 bits wide. Since the 8051 has an 8-bit architecture, each 16-bit timer is accessed as two separate registers of low byte and high byte. Each timer is discussed separately.

Timer 0 registers

The 16-bit register of Timer 0 is accessed as low byte and high byte. The low byte register is called TL0 (Timer 0 low byte) and the high byte register is referred to as TH0 (Timer 0 high byte).



These registers can be accessed like any other register, such as A, B, RO, Rl, R2, etc.

For example, the instruction “MOV TL0 , #4FH” moves the value 4FH into TL0,

Timer 1 Registers

Timer I is also 16 bits, and its 16-bit register is split into two bytes, referred to as TLl (Timer I low byte) and TH1 (Timer 1 high byte). These registers are accessible in the same way as the registers of Timer 0.

TMOD (timer mode) register

Both timers 0 and 1 use the same register, called TMOD, to set the various timer operation modes. TMOD is an 8-bit register in which the lower 4 bits are used for Timer 0 and the upper 4 bits for Timer 1. In each case, the lower 2 bits are used to set the timer mode and the upper 2 bits to specify the operation. The Bit format of TMOD register is as shown below.

M1, MO

M0and M 1 selects the timer mode. There are three modes: 0, 1, and 2.

- Mode 0 is a 13-bit timer,
- Mode 1 is a 16-bit timer, and
- Mode 2 is an 8-bit timer.

(MSB) (LSB)

GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer 1				Timer 0			

GATE Gating control when set. The timer/counter is enabled only while the INTx pin is high and the TRx control pin is set. When cleared, the timer is enabled whenever the TRx control bit is set.

C/T Timer or counter selected cleared for timer operation (input from internal system clock). Set for counter operation (input from Tx input pin).

M1 Mode bit 1

M0 Mode bit 0

M1	M0	Mode	Operating Mode
0	0	0	13-bit timer mode 8-bit timer/counter THx with TLx as 5-bit prescaler
0	1	1	16-bit timer mode 16-bit timer/counters THx and TLx are cascaded; there is no prescaler
1	0	2	8-bit auto reload 8-bit auto reload timer/counter; THx holds a value that is to be reloaded into TLx each time it overflows.
1	1	3	Split timer mode

C/T (clock/timer)

This bit in the TMOD register is used to decide whether the timer is used as a delay generator or an event counter. If C/T = 0, it is used as a timer for time delay generation. The clock source for the time delay is the crystal frequency of the 8051..

Example -1

Indicate which mode and which timer are selected for each of the following.

(a) MOV TMOD,#01H (b) MOV TMOD,#20H (c) MOV TMOD,#12H

Solution:

We convert the values from hex to binary. From Figure 9-3 we have:

1. TMOD = 00000001, mode 1 of Timer 0 *is* selected.
2. TMOD = 00100000, mode 2 of Timer 1 is selected.
    1. TMOD = 00010010, mode 2 of Timer 0, and mode 1 of Timer 1 are selected.

Clock source for timer

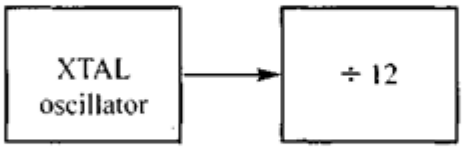
The crystal frequency attached to the 8051 is the source of the clock for the timer. This means that the crystal frequency attached to the 8051 decides the speed at which the 8051 timer ticks. The frequency for the timer is always 1 / 12th the frequency of the crystal attached to the 8051.

Example -2

Find the timer's clock frequency and its period for various 8051-based systems. with the following crystal frequencies.

(a) 12 MHz
(b) 16 MHz
(c) 11.0592 MHz

Solution:



(a) 1/12 × 12 MHz = 1 MHz and T = 1/1 MHz = 1 µs

(b) 1/12 × 16 MHz = 1.333 MHz and T = 1/1.333 MHz = .75 µs

(c) 1/12 × 11.0592 MHz = 921.6 kHz;
T = 1/921.6 kHz = 1.085 µs

GATE

The other bit of the TMOD register is the GATE bit. Every timer has a means of starting and stopping. Some timers do this by software, some by hardware, and some have both software and hardware controls.

The start and stop of the timer are controlled by way of software by the TR (timer start) bits TR0 and TR1. This is achieved by the instructions "SETB TR1″ and "CLR TR1″ for Timer 1, and "SETB TR0" and "CLR TR0" for Timer 0. The SETB instruction starts it, and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register.

The hardware way of starting and stopping the timer by an external source is achieved by making GATE = 1 in the TMOD register.

Example - 3

Find the value for TMOD if we want to program Timer 0 in mode 2, use 8051 XTAL for the clock source, and use instructions to start and stop the timer.

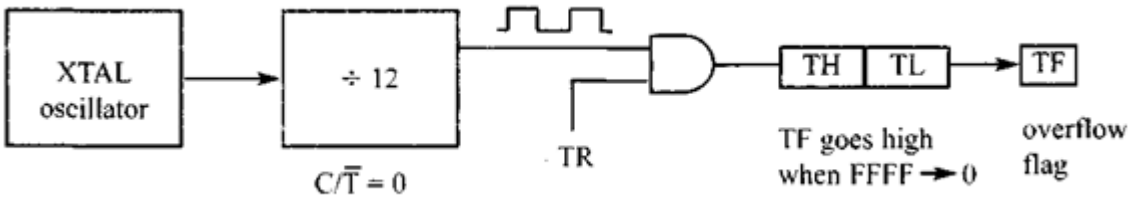
Solution:

TMOD= 0000 0010 Timer 0, mode 2,
C/T = 0 to use XTAL clock source, and
gate = 0 to use internal (software)
start and stop method.

Mode 1 programming

The following are the characteristics and operations of mode 1:

1. It is a 16-bit timer; therefore, it allows values of 0000 to FFFFH to be loaded into the timer's registers TL and TH.
2. After TH and TL are loaded with a 16-bit initial value, the timer must be started. This is done by "SETB TR0" for Timer 0 and "SETB TR1″ for Timer 1.
3. After the timer is started, it starts to count up. It counts up until it reaches its limit of FFFFH. When it rolls over from FFFFH to 0000, it sets high a timer flag. This timer flag can be monitored. When this timer flag is raised, one option would be to stop the timer with the instructions "CLR TR0" or "CLR TR1″, for Timer 0 and Timer 1, respectively.
4. After the timer reaches its limit and rolls over, in order to repeat the process the registers TH and TL must be reloaded with the original value, and TF must be reset to 0.



Steps to Program in mode 1

To generate a time delay, using the timer's mode 1, the following steps are taken.

1. Load the TMOD value register indicating which timer (Timer 0 or Timer 1) is to be used and which timer mode (0 or 1) is selected.
2. Load registers TL and TH with initial count values.
3. Start the timer.
4. Keep monitoring the timer flag (TF) with the "JNB TFx, target" instruction to see if it is raised. Get out of the loop when TF becomes high.
5. Stop the timer.
6. Clear the TF flag for the next round.
7. Go back to Step 2 to load TH and TL again.

Example -4

In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the PI.5 bit. Timer 0 is used to generate the time delay. Analyze the program.

```
            MOV    TMOD,#01        ;Timer 0, mode 1(16-bit mode)
HERE:       MOV    TL0,#0F2H       ;TL0 = F2H, the Low byte
            MOV    TH0,#0FFH       ;TH0 = FFH, the High byte
            CPL    P1.5            ;toggle P1.5
            ACALL  DELAY
            SJMP   HERE            ;load TH, TL again
;----------delay using Timer 0
DELAY:
            SETB   TR0             ;start Timer 0
AGAIN:      JNB    TF0,AGAIN       ;monitor Timer 0 flag until
                                   ;it rolls over
            CLR    TR0             ;stop Timer 0
            CLR    TF0             ;clear Timer 0 flag
            RET
```

Solution:

In the above program notice the following steps.

1. TMOD is loaded.
2. FFF2H is loaded into THO – TLO.
3. P1.5 is toggled for the high and low portions of the pulse.
4. The DELAY subroutine using the timer is called.
5. In the DELAY subroutine, Timer 0 is started by the "SETB TRO" instruction.
6. Timer 0 counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of FFF3, FFF4, FFF5,FFF6, FFF7, FFF8, FFF9, FFFA, FFFB, and so on until it reaches FFFFH. One more clock rolls it to 0, raising the timer flag (TFO = 1). At that point, the JNB instruction falls through.
7. Timer 0 is stopped by the instruction "CLR TRO". The DELAY subroutine ends, and the process is repeated.

To repeat the process, we must reload the TL and TH registers and start the timer again.

Example -5

In Example 4, calculate the amount of time delay in the DELAY subroutine generated by the timer. Assume that XTAL = 11.0592 MHz.

Solution:

- The timer works with a clock frequency of 1/12 of the XTAL frequency;
- Therefore, we have 11.0592 MHz / 12 = 921.6 kHz as the timer frequency.
- As a result, each clock has a period of T = 1 / 921.6 kHz = 1.085 (is Timer 0 counts up each 1.085 us resulting in delay = number of counts x 1.085 us.
- The number of counts for the rollover is FFFFH – FFF2H = ODH (13 decimal). However, we add one to 13 because of the extra clock needed when it rolls over from FFFF to 0 and raises the TF flag. This gives 14 x 1.085 us = 15.19 us for half the pulse. For the entire period T = 2 x 15.19 (as = 30.38 (is gives us the time delay generated by the timer.

Example -6

In Example 6, calculate the frequency of the square wave generated on pin PI. 5.

Solution:

In the time delay calculation of Example -5, we did not include the overhead due to instructions in the loop. To get a more accurate timing, we need to add clock cycles due to the instructions in the loop.

```
                                          Cycles
HERE:     MOV   TL0,#0F2H                   2
          MOV   TH0,#0FFH                   2
          CPL   P1.5                        1
          ACALL DELAY                       2
          SJMP  HERE                        2
;---------delay using Timer 0
DELAY:
          SETB  TR0                         1
AGAIN:    JNB   TF0,AGAIN                   14
          CLR   TR0                         1
          CLR   TF0                         1
          RET                               2
                                Total       28
```

T = 2 × 28 × 1.085 µs = 60.76 µs and F = 16458.2 Hz.

Example -7

Find the delay generated by Timer 0 in the following code, using both of the methods. Do not include the overhead due to instructions.

```
        CLR   P2.3          ;clear P2.3
        MOV   TMOD,#01      ;Timer 0, mode 1(16-bit mode)
HERE:   MOV   TL0,#3EH      ;TL0 = 3EH, Low byte
        MOV   TH0,#0B8H     ;TH0 = B8H, High byte
        SETB  P2.3          ;SET high P2.3
        SETB  TR0           ;start Timer 0
AGAIN:  JNB   TF0,AGAIN     ;monitor Timer 0 flag
        CLR   TR0           ;stop Timer 0
        CLR   TF0           ;clear Timer 0 flag for
                            ;next round
        CLR   P2.3
```

Solution:

1. (FFFF-B83E + 1) = 47C2H= 18370 in decimal and 18370 x 1.085 fis= 19.93145 ms.
2. Since TH – TL = B83EH = 47166 (in decimal) we have 65536 – 47166 = 18370. This means that the timer counts from B83EH to FFFFH.. This plus rolling over to 0 goes through a total of 18370 clock cycles, where each clock is 1.085 *ls* in duration. Therefore, we have 18370 x 1.085 (is = 19.93145 ms as the width of the pulse.

Example -8

The following program generates a square wave on pin PL5 continuously using Timer 1 for a time delay. Find the frequency of the square wave if XTAL =11.0592 MHz. In your calculation do not include the overhead due to instructions in the loop.

```
        MOV   TMOD,#10H     ;Timer 1, mode 1(16-bit)
AGAIN:  MOV   TL1,#34H      ;TL1 = 34H, Low byte
        MOV   TH1,#76H      ;TH1 = 76H, High byte
                            ;(7634H = timer value)
        SETB  TR1           ;start Timer 1
BACK:   JNB   TF1,BACK      ;stay until timer rolls over
        CLR   TR1           ;stop Timer 1
        CPL   P1.5          ;comp. P1.5 to get hi, lo
        CLR   TF1           ;clear Timer 1 flag
        SJMP  AGAIN         ;reload timer since Mode 1
                            ;is not auto-reload
```

Solution:

In the above program notice the target of SJMP. In mode 1, the program must reload the TH, TL register every time if we want to have a continuous wave. Now the calculation. Since FFFFH – 7634H = 89CBH + 1 = 89CCH and 89CCH = 35276 clock count. 35276 x 1.085 us = 38.274 ms for half of the square wave. The entire square wave length is 38.274 x 2 = 76.548 ms and has a frequency = 13.064 Hz.

Also notice that the high and low portions of the square wave pulse are equal. In the above calculation, the overhead due to all the instructions in the loop is not included.

Finding values to be loaded into the timer

To calculate the values to be loaded into the TL and TH registers

Assuming XTAL = 11.0592 MHz from we can use the following steps for finding the TH, TL registers' values.

1. Divide the desired time delay by 1.085 us.
2. Perform 65536 – *n,* where *n* is the decimal value we got in Step 1.
3. Convert the result of Step 2 to hex, where *yyxx* is the initial hex value to be loaded into the timer's registers.
4. Set TL = *xx* and TH = *yy*

Example -9

Assume that XTAL = 11.0592 MHz. What value do we need to load into the timer's registers if we want to have a time delay of 5 ms (milliseconds)? Show the program for Timer 0 to create a pulse width of 5 ms on P2.3.

```
        CLR   P2.3          ;clear P2.3
        MOV   TMOD,#01      ;Timer 0, mode 1 (16-bit mode)
HERE:   MOV   TL0,#0        ;TL0 = 0, Low byte
        MOV   TH0,#0EEH     ;TH0 = EE( hex), High byte
        SETB  P2.3          ;SET P2.3 high
        SETB  TR0           ;start Timer 0
AGAIN:  JNB   TF0,AGAIN     ;monitor Timer 0 flag
                            ;until it rolls over
        CLR   P2.3          ;clear P2.3
        CLR   TR0           ;stop Timer 0
        CLR   TF0           ;clear Timer 0 flag
```

Example -10

Assuming that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 kHz frequency on pin PI .5.

Solution:

This is similar to Example 9-10, except that we must toggle the bit to generate the square wave. Look at the following steps.

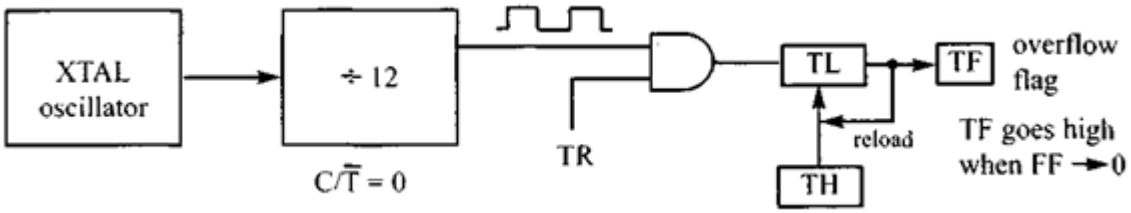
1. T = 1 / f = 1 / 2 kHz = 500 us the period of the square wave.
2. 1/2 of it for the high and low portions of the pulse is 250 us.
3. 250 us / 1.085 us = 230 and 65536 – 230 = 65306. which in hex is FF1AH.
4. TL = 1AH and TH = FFH. all in hex. The program is as follows.

```
        MOV   TMOD,#10H       ;Timer 1, mode 1(16-bit)
AGAIN:  MOV   TL1,#1AH        ;TL1=1AH, Low byte
        MOV   TH1,#0FFH       ;TH1=FFH, High byte
        SETB  TR1             ;start Timer 1
BACK:   JNB   TF1,BACK        ;stay until timer rolls over
        CLR   TR1             ;stop Timer 1
        CPL   P1.5            ;complement P1.5 to get hi, lo
        CLR   TF1             ;clear Timer 1 flag
        SJMP  AGAIN           ;reload timer since mode 1
                              ;is not auto-reload
```

Mode 2 programming

The following are the characteristics and operations of mode 2.

1. It is an 8-bit timer; therefore, it allows only values of 00 to FFH to be loaded into the timer's register TH.
2. After TH is loaded with the 8-bit value, the 8051 gives a copy of it to TL. Then the timer must be started. This is done by the instruction "SETB TRO" for Timer 0 and "SETB TR1[1]" for Timer 1. This is just like mode 1.
3. After the timer is started, it starts to count up by incrementing the TL registers. It counts up until it reaches its limit of FFH. When it rolls over from FFH to 00, it sets high the TF (timer flag). If we are using Timer 0, TFO goes high; if we are using Timer 1, TF1 is raise



4. When the TL register rolls from FFH to 0 and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register. To repeat the process, we must simply clear TF and let it go without any need by the programmer to reload the original value. This makes mode 2 an auto-reload, in contrast with mode 1 in which the programmer has to reload TH and TL

It must be emphasized that mode 2 is an 8-bit timer. However, it has an auto-reloading capability. In auto-reload, TH is loaded with the initial count and a copy of it is given to TL.

Steps to program in mode 2

1. To generate a time delay using the timer's mode 2, take the following steps.
2. Load the TMOD value register indicating which timer (Timer 0 or Timer 1) is to be used, and select the timer mode (mode 2).

3. Load the TH registers with the initial count value.
4. Start the timer.
5. Keep monitoring the timer flag (TF) with the "JNB TFx, target" instruction to see whether it is raised. Get out of the loop when TF goes high.
6. Clear the TF flag.
7. Go back to Step 4, since mode 2 is auto-reload

Example 11

Assuming that XTAL = 11.0592 MHz. find (a) the frequency of the square wave generated on pin P 1.0 in the following program, and (b) the smallest frequency achievable in this program, and the TH value to do that.

```
        MOV   TMOD,#20H       ;T1/mode 2/8-bit/auto-reload
        MOV   TH1,#5          ;TH1 = 5
        SETB  TR1             ;start Timer 1
BACK:   JNB   TF1,BACK        ;stay until timer rolls over
        CPL   P1.0            ;comp. P1.0 to get hi, lo
        CLR   TF1             ;clear Timer 1 flag
        SJMP  BACK            ;mode 2 is auto-reload
```

Solution:

1. First notice the target address of SJMP. In mode 2 we do not need to reload TH since it is auto-reload. Now (256 – 05) x 1.085 us = 251 x 1.085 us = 272.33 us is the high portion of the pulse. Since it is a 50% duty cycle square wave, the period T is twice that; as a result T = 2 x 272.33 us = 544.67 us and the frequency = 1.83597 kHz.
2. To get the smallest frequency, we need the largest T and that is achieved when TH = 00. In that case, we have T = 2 x 256 x 1.085 us = 555.52 us and the frequency = 1.8 kHz.

Program -12

Find the frequency of a square wave generated on pin P1.0.

Solution:

```
        MOV   TMOD,#2H        ;Timer 0, mode 2
                              ;(8-bit, auto-reload)
        MOV   TH0,#0          ;TH0=0
AGAIN:  MOV   R5,#250         ;count for multiple delay
        ACALL DELAY
        CPL   P1.0            ;toggle P1.0
        SJMP  AGAIN           ;repeat
DELAY:  SETB  TR0             ;start Timer 0
BACK:   JNB   TF0,BACK        ;stay until timer rolls over
        CLR   TR0             ;stop Timer 0
        CLR   TF0             ;clear TF for next round
        DJNZ  R5,DELAY
        RET
```

T = 2 (250 × 256 × 1.085 μs) = 138.88 ms, and frequency = 72 Hz.