MODULE 3: 8051 Programming in C

Structure

3 a)

3.1 Data types and time delay in 8051C

3.2 IO programming in 8051C

3.3 Logic operations in 8051 C

3.4 Data conversion program in 8051 C

3.5 Accessing code ROM space in 8051C

3.6 Data serialization using 8051C

3 b) 8051 Timer programming in Assembly and C:

3.7 Programming 8051 timers

3.8 Counter programming

3.9 Programming timers 0 and 1 in 8051 C

Objectives

- To explain in detail the execution of 8051 C language
- To explain develop 8051C programs for time delay

3.1 Data types in 8051C

Why program the 8051 in C?

Compilers produce hex files that we download into the ROM of the microcontroller. The size of the hex file produced by the compiler is one of the main concerns of microcontroller programmers, for two reasons:

1. Microcontrollers have limited on-chip ROM.
2. The code space for the 8051 is limited to 64K bytes.

Following are some of the major reasons for writing programs in C instead of Assembly:

1. It is easier and less time consuming to write in C than Assembly.
2. C is easier to modify and update.
3. You can use code available in function libraries.
4. C code is portable to other microcontrollers with little or no modification.

C data types for the 8051

Since one of the goals of 8051 C programmers is to create smaller hex files, it is worthwhile to re-examine C data types for 8051 C. In other words, a good understanding of C data types for the 8051 can help programmers to create smaller hex files. In this section we focus on the specific C data types that are most useful and widely used for the 8051 microcontroller.

1. **Unsigned char**

A. Since the 8051 is an 8-bit microcontroller, the character data type is the most natural choice for many applications. The unsigned char is an 8-bit data type that takes a value in the range of 0 – 255 (00 – FFH). It is one of the most widely used data types for the 8051. In many situations, such as setting a counter value.

B. Where there is no need for signed data we should use the unsigned char instead of the signed char. Remember that C compilers use the signed char as the default if we do not put the keyword *unsigned* in front of the char (see Example 1-1). We can also use the unsigned char data type for a string of ASCII characters, including extended

ASCII characters. Example 1-2 shows a string of ASCII characters. See Example 1-3 for toggling ports.

C. In declaring variables, we must pay careful attention to the size of the data and try to use unsigned char instead of int if possible. Because the 8051 has a limited number of registers and data RAM locations, using the int in place of the char data type can lead to a larger size hex file. Such a misuse of the data types in compilers such as Microsoft Visual C++ for x86 IBM PCs is not a significant issue.

Example 1-1

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;
  }
```

Run the above program on your simulator to see how P1 displays values 00 - FFH in binary.

Example 1-2

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[ ] = “0,1,2,3,4,5,A,B,C,D”;
Unsigned char z;
For(z=0;z<=10;z++)
P1=mynum(z);
}
```

Example 1-3

Write an 8051 C program to toggle all the bits of PI continuously.

Solution:

// Toggle PI forever

include <reg51.h>

void main(void)

```
{
  for(;;)        //repeat forever
   {
     P1=0x55;    //0x indicates the data is in hex (binary)
     P1=0xAA;
   }
}
```

2. Signed char

The signed char is an 8-bit data type that uses the most significant bit (D7 of D7 – DO) to represent the – or + value. As a result, we have only 7 bits for the magnitude of the signed number, giving us values from -128 to +127. In situations where + and – are needed to represent a given quantity such as temperature, the use of the signed char data type is a must. Again notice that if we do not use the keyword *unsigned,* the default is the signed value. For that reason we should stick with the unsigned char unless the data needs to be represented as signed numbers.

Example 1-4

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];
  }
```

3. Unsigned int

1. The unsigned int is a 16-bit data type that takes a value in the range of 0 to 65535 (0000 – FFFFH). In the 8051, unsigned int is used to define 16-bit variables such as memory addresses. It is also used to set counter values of more than 256.
2. Since the 8051 is an 8-bit microcontroller and the int data type takes two bytes of RAM, we must not use the int data type unless we have to.
3. Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file. Such misuse is not a big deal in PCs with 256 megabytes of memory, 32-bit Pentium registers and memory accesses, and a bus speed of 133 MHz.
4. However, for 8051 programming do not use unsigned int in places where unsigned char will do the job. Of course the compiler will not generate an error for this misuse, but the overhead in hex file size is noticeable.
5. Also in situations where there is no need for signed data (such as setting counter values), we should use unsigned int instead of signed int.
6. This gives a much wider range for data declaration. Again, remember that the C compiler uses signed int as the default if we do not use the keyword *unsigned.*

4. Signed int

Signed int is a 16-bit data type that uses the most significant bit (015 of D15 – DO) to represent the – or + value. As a result, we have only 15 bits for the magnitude of the number, or values from -32,768 to +32,767.

5. Sbit (single bit)

The sbit keyword is a widely used 8051 C data type designed specifically to access single-bit addressable registers. It allows access to the single bits of the SFR registers. We can use sbit to access the individual bits of the ports as shown in Example 1-5.

Example 1-5

Write an 8051 C program to toggle bit DO of the port P1 (Pl.0) 50,000 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;
      }
  }
```

Bit and sfr

The bit data type allows access to single bits of bit-addressable memory spaces 20 – 2FH. Notice that while the sbit data type is used for bit-addressable SFRs, the bit data type is used for the bit-addressable section of RAM space 20 -2FH. To access the byte-size SFR registers, we use the sfr data type. We will see the use of sbit, bit, and sfr data types in the next section.

Table 3.1:Some Widely Used Data Types for 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

3.1.1 Time delay in C

There are two ways to create a time delay in 8051 C:

1. Using a simple for loop
2. Using the 8051 timers

In either case, when we write a time delay we must use the oscilloscope to measure the duration of our time delay. Next, we use the for loop to create time delays.

In creating a time delay using a for loop, we must be mindful of three factors that can affect the accuracy of the delay.

1. The 8051 design. Since the original 8051 was designed in 1980, both the fields of 1C technology and microprocessor architectural design have seen great advancements.The number of machine cycles and the number of clock periods per machine cycle vary among different versions of the 8051/52 microcontroller.
2. 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.
   For example, the DS5000 uses 4 clock periods per machine cycle, while the DS89C420 uses only one clock per machine cycle.
3. 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.
4. Compiler choice. The third factor that affects the time delay is the compiler used to compile the C program. When we program in Assembly language, we can control the exact instructions and their sequences used in the delay sub routine. 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, dif ferent compilers produce different code. In other words, if we compile a given 8051 C programs with different compilers, each compiler produces different hex code.

Example 1-6

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++);
    }
  }
```

Example 1-7

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

Solution:

The program below is tested for the DS89C420 with XTAL = 11.0592 MHz.

```
#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++);
  }
```

Example 1-8

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++);
  }
```

3.2 IO programming in 8051C

Byte size I/O

As we stated in Chapter 4, ports PO – P3 are byte-accessible. We use the PO – P3 labels as defined in the 8051/52 C header file.

Example 1-9

LEDs are connected to bits PI and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

Solution:

```
#include <reg51.h>
#define LED P2          //notice how we can define P2
void main(void)
  {
    P1=00;              //clear P1
    LED=0;              //clear P2
    for(;;)             //repeat forever
      {
        P1++;           //increment P1
        LED++;          //increment P2
      }
  }
```

Example 1-10

Write an 8051 C program to get a byte of data from PI, wait 1/2 second, and then send it to P2.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    unsigned char mybyte;
    P1=0xFF;                //make P1 an input port
    while(1)
      {
        mybyte=P1;          //get a byte from P1
        MSDelay(500);
        P2=mybyte;          //send it to P2
      }
  }

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

Example 1-11

Write an 8051 C program to get a byte of data from PO. If it is less than 100, send it to PI; otherwise, send it to P2.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mybyte;
    P0=0xFF;                //make P0 an input port
    while(1)
      {
        mybyte=P0;          //get a byte from P0
        if(mybyte<100)
          P1=mybyte;        //send it to P1 if less than 100
        else
          P2=mybyte;        //send it to P2 if more than 100
      }
  }
```

3.2.1 Bit-addressable I/O programming

1. The I/O ports of PO – P3 are bit-addressable. We can access a single bit without disturbing the rest of the port. We use the sbit data type to access a single bit of PO – P3.One way to do that is to use the Px^y format where *x* is the port 0, 1, 2, or 3, and *y* is the bit 0 – 7 of that port.
2. For example, P1^7 indicates PI.7. When using this method, you need to include the reg51 .h file. Study the next few examples to become familiar with the syntax.

Example 1-12

Write an 8051 C program to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2.

Solution:

```
//toggling an individal bit
#include <reg51.h>
sbit mybit = P2^4;    //notice the way single bit is declared
void main(void)
{
  while(1)
    {
      mybit=1;        //turn on P2.4
      mybit=0;        //turn off P2.4
    }
  }
```

Example 1-13

Write an 8051 C program to monitor bit PI.5. If it is high, send 55H to PO; otherwise, sendAAHtoP2.

Solution:

```
#include <reg51.h>
sbit mybit = P1^5;    //notice the way single bit is declared
void main(void)
  {
    mybit=1;          //make mybit an input
    while(1)
      {
        if(mybit==1)
          P0=0x55;
        else
          P2=0xAA;
      }
  }
```

Example 1-14

A door sensor is connected to the P 1.1 pin, and a buzzer is connected to PI.7. Write an 8051 C program to monitor the door sensor, and when it opens, sound the buzzer. You can sound the buzzer by sending a square wave of a few hundred Hz.

Solution:

```
#include <reg51.h>
void MSDelay(unsigned int);
sbit Dsensor = P1^1; //notice the way single bit is defined
sbit Buzzer = P1^7;
void main(void)
  {
    Dsensor=1;        //make P1.1 an input
    while(Dsensor==1)
      {
        buzzer=0;
        MSDelay(200);
        buzzer=1;
        MSDelay(200);
      }
  }

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

Example 1-15

The data pins of an LCD are connected to PI. The information is latched into the LCD whenever its Enable pin goes from high to low. Write an 8051 C program to send "The Earth is but One Country" to this LCD.

Solution:

```
#include <reg51.h>
#define LCDData P1          //LCDData declaration
sbit En=P2^0;               //the enable pin
void main(void)
  {
    unsigned char message[ ]= "The Earth is but One Country";
    unsigned char z;
    for(z=0;z<28;z++)       //send all the 28 characters
      {
       LCDData=message[z];
       En=1;                //a high-
       En=0;                //-to-low pulse to latch the LCD data
      }
  }
```

3.2.2 Accessing SFR addresses 80 – FFH

Another way to access the SFR RAM space 80 – FFH is to use the sfr data type. We can also access a single bit of any SFR if we specify the bit address as shown in Example 1-16. Both the bit and byte addresses for the PO – P3 ports are given in Table 1.2. Notice in Examples 1- 16, that there is no ^include <reg51.h> statement. This allows us to access any byte of the SFR RAM space 80 – FFH. This is a method widely used for the new generation of 8051 microcontrollers, and we will use it in future chapters.

Table **2:** Single Bit Addresses of Ports

P0	Addr	P1	Addr	P2	Addr	P3	Addr	Port's Bit
P0.0	80H	P1.0	90H	P2.0	A0H	P3.0	B0H	D0
P0.1	81H	P1.1	91H	P2.1	A1H	P3.1	B1H	D1
P0.2	82H	P1.2	92H	P2.2	A2H	P3.2	B2H	D2
P0.3	83H	P1.3	93H	P2.3	A3H	P3.3	B3H	D3
P0.4	84H	P1.4	94H	P2.4	A4H	P3.4	B4H	D4
P0.5	85H	P1.5	95H	P2.5	A5H	P3.5	B5H	D5
P0.6	86H	P1.6	96H	P2.6	A6H	P3.6	B6H	D6
P0.7	87H	P1.7	97H	P2.7	A7H	P3.7	B7H	D7

Example 1-16

Write an 8051 C program to get the status of bit Pl.0, save it, and send it to P2.7 continuously.

Solution:

Write an 8051 C program to turn bit P1.5 on and off 50,000 times.
Solution:

```
sbit MYBIT = 0x95;  //another way to declare bit P1^5
void main(void)
  {
    unsigned int z;
    for(z=0;z<50000;z++)
      {
        MYBIT=1;
        MYBIT=0;
      }
  }
```

Example 1-17

Write an 8051 C program to toggle all the bits of PO, P1, and P2 continuously with a 250 ms delay.' Use the sfr keyword to declare the port addresses.

```
// Accessing Ports as SFRs using the sfr data type
sfr P0 = 0x80;          //declaring P0 using sfr data type
sfr P1 = 0x90;
sfr P2 = 0xA0;
void MSDelay(unsigned int);
void main(void)
  {
    while(1)            //do it forever
      {
        P0=0x55;
        P1=0x55;
        P2=0x55;
        MSDelay(250);   //250 ms delay
        P0=0xAA;
        P1=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++);
  }
```

Using bit data type for bit-addressable RAM

The sbit data type is used for bit-addressable SFR registers only. Sometimes we need to store some data in a bit-addressable section of the data RAM space 20 – 2FH. To do that, we use the bit data type, as shown in Example 1-18.

Example 1-18

Write an 8051 C program to get the status of bit Pl.O, save it, and send it to P2.7 continuously.

Solution:

```
#include <reg51.h>
sbit inbit = P1^0;
sbit outbit = P2^7;       //sbit is used to declare SFR bits
bit membit;               //notice we use bit to declare
                          //bit-addressable memory
void main(void)
  {
    while(1)
      {
        membit=inbit;     //get a bit from P1.0
        outbit=membit;    //and send it to P2.7
      }
  }
```

3.3 Logic operations in 8051 C

Bit-wise operators in C

1. While every C programmer is familiar with the logical operators AND (&&), OR (||), and NOT (!), many C programmers are less familiar with the bitwise operators AND (&), OR (|), EX-OR (^), Inverter (~), Shift Right (»), and Shift Left («).
2. These bit-wise operators are widely used in software engineering for embedded systems and control; consequently, understanding and mastery of them are critical in microprocessor-based system design and interfacing. See Table 1-3.

Table 1-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

The following shows some examples using the C !ogical operators.

```
1. 0x35 & 0x0F = 0x05      /* ANDing */
2. 0x04 | 0x68 = 0x6C      /* ORing:  */
3. 0x54 ^ 0x78 = 0x2C      /* XORing */
4. ~0x55 = 0xAA            /* Inverting 55H */
```

Example 1-19

Run the following program on your simulator and examine the results.

Solution:

```
#include <reg51.h> void main (void)
{
P0 = 0×35 & OxOF; //ANDing
Pl = 0×04 | 0×68; //ORing
P2= 0×54 * 0×78; //XORing
P0= -0×55; //inversing
Pl= Ox9A » 3; //shifting right 3 times
P2= 0×77 » 4; //shifting right 4 times
p0= 0×6 « 4; //shifting left 4 times
```

Example 1-20

Write an 8051 C program to toggle all the bits of PO and P2 continuously with a 250 ms delay. Use the inverting operator.

Solution:

The program below is tested for the DS89C420 with XTAL = 11.0592 MHz.

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    P0=0x55;
    P2=0x55;
    while(1)
      {
        P0=~P0;
        P2=~P2;
        MSDelay(250);
      }
  }

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

3.3.1 Bit-wise shift operation in C

There are two bit-wise shift operators in C: (1) shift right (»), and (2) shift left («).

Their format in C is as follows:

data » number of bits to be shifted right

data « number of bits to be shifted left

The following shows some examples of shift operators in C.

```
1. 0x9A >> 3= 0x13        /* shifting right 3 times */
2. 0x77 >> 4 = 0x07       /* shifting right 4 times */
3. 0x6 << 4 = 0x60        /* shifting left 4 times */
```

Example 1-21

Write an 8051 C program to toggle all the bits of PO, PI, and P2 continuously with a 250 ms delay. Use the Ex-OR operator.

Solution:

The program below is tested for the DS89C420 with XTAL = 11.0592 MHz.

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
  {
    P0=0x55;
    P1=0x55;
    P2=0x55;
    while(1)
      {
        P0=P0^0xFF;
        P1=P1^0xFF;
        P2=P2^0xFF;
        MSDelay(250);
      }
  }

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

Example 1-22

Write an 8051 C program to get bit Pl.0 and send it to P2.7 after inverting it.

Solution:

```
#include <reg51.h>
sbit inbit=P1^0;
sbit outbit=P2^7;      //sbit is used declare port (SFR) bits
bit membit;            //notice this is bit-addressable memory
void main(void)
  {
    while(1)
      {
        membit=inbit;      //get a bit from P1.0
        outbit=~membit;    //invert it and send it to P2.7
      }
  }
```

Example 1-23

Write an 8051 C program to read the P1.0 and P1.1 bits and issue an ASCII character to PO according to the following table.

P1.1	P1.0	
0	0	send '0' to P0
0	1	send '1' to P0
1	0	send '2' to P0
1	1	send '3' to P0

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char z;
    z=P1;                    //read P1
    z=z&0x3;                 //mask the unused bits
    switch(z)                //make decision
      {
        case(0):
         {
           P0='0';           //issue ASCII 0
           break;
         }
        case(1):
         {
           P0='1';           //issue ASCII 1
           break;
         }
        case(2):
         {
           P0='2';           //issue ASCII 2
           break;
         }
        case(3):
         {
           P0='3';           //issue ASCII 3
           break;
         }
      }
  }
```

3.4 Data conversion program in 8051 C

a)ASCII numbers

On ASCII keyboards, when the key "0″ is activated, "Oil 0000″ (30H) is provided to the computer. Similarly, 31H (Oil 0001) is provided for the key "1″, and so on, as shown in Table 1-4.

Table 1-4: 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

b) Packed BCD to ASCII conversion

The RTC 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 Oil 0000 (30H). The following demonstrates converting from packed BCD to ASCII. See also Example 7-24.

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

c) 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 Chapter 16**discusses** the RTC chip and uses the BCD and ASCII conversion programs shown in Examples 1-24 and 1-25.

Example 1-24

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
  }
```

Example 7-25

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;
  }
```

3.4.1 Checksum byte in ROM

1. To ensure the integrity of ROM contents, every system must perform the checksum calculation.
2. The process of checksum will detect any corruption of the contents of ROM. One of the causes of ROM corruption is current surge, either when the system is turned on or during operation.
3. To ensure data integrity in ROM, the checksum process uses what is called a ***checksum byte.***

4. The checksum byte is an extra byte that is tagged to the end of a series of bytes of data. To calculate the checksum byte of a series of bytes of data, the following steps can be taken.
5. Add the bytes together and drop the carries.
    1. Take the 2′s complement of the total sum. This is the checksum byte, which becomes the last byte of the series.
    2. To perform the checksum operation, add all the bytes, including the checksum byte. The result must be zero. If it is not zero, one or more bytes of data have been changed (corrupted).

Example 1-26

Assume that we have 4 bytes of hexadecimal data: 25H, 62H, 3FH, and 52H. (a) Find the checksum byte, (b) perform the checksum operation to ensure data integrity, and (c) if the second byte 62H has been changed to 22H, show how checksum detects the error.

Solution:

(a) Find the checksum byte.

```
   25H
+  62H
+  3FH
+  52H
  118H  (Dropping carry of 1 and taking the 2's complement, we get E8H.)
```

(b) Perform the checksum operation to ensure data integrity.

```
   25H
+  62H
+  3FH
+  52H
+  E8H
  200H  (Dropping the carries we get 00, which means data is not corrupted.)
```

(c) If the second byte 62H has been changed to 22H, show how checksum detects the error.

```
   25H
+  22H
+  3FH
+  52H
+  E8H
  1C0H  (Dropping the carry, we get C0H, which means data is corrupted.)
```

Example 1-27

Write an 8051 C program to calculate the checksum byte for the data given in Example 7-26.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mydata[] = {0x25,0x62,0x3F,0x52};
    unsigned char sum=0;
    unsigned char x;
    unsigned char chksumbyte;
    for(x=0;x<4;x++)
      {
        P2=mydata[x];            //issue each byte to P2
        sum=sum+mydata[x];       //add them together
        P1=sum;                  //issue the sum to P1
      }
    chksumbyte=~sum+1;           //make 2's complement
    P1=chksumbyte;               //show the checksum byte
  }
```

Example 1-28

Write an 8051 C program to perform step (b) of Example 7-26. If data is good, send ASCII character 'G' to PO. Otherwise send 'B' to PO.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mydata[]={0x25,0x62,0x3F,0x52,0xE8};
    unsigned char chksum=0;
    unsigned char x;
    for(x=0;x<5;x++)
      chksum=chksum+mydata[x];   //add them together
    if(chksum==0)
      P0='G';
    else
      P0='B';
  }
```

3.4.2 Binary (hex) to decimal and ASCII conversion in 8051 C

1. The printf function is part of the standard I/O library in C and can do many things, including converting data from binary (hex) to decimal, or vice versa. But printf takes a lot of memory space and increases your hex file substantially. For this reason, in systems based on the 8051 microcontroller, it is better to write your own conversion function instead of using printf.
2. One of the most widely used conversions is the binary to decimal conversion. In devices such as ADC (Analog-to-Digital Conversion) chips, the data is provided to the microcontroller in binary.
3. In some RTCs, data such as time and dates are also provided in binary. In order to display binary data we need to convert it to decimal and then to ASCII. Since the hexadecimal format is a convenient way of representing binary data we refer to the binary data as hex.
4. The binary data 00 – FFH converted to decimal will give us 000 to 255. One way to do that is to divide it by 10 and keep the remainder. For example, 11111101 or FDH is 253 in decimal. The following is one version of an algorithm for conversion of hex (binary) to decimal:

	Quotient	Remainder
FD/0A	19	3 (low digit) LSD
19/0A	2	5 (middle digit)
		2 (high digit) (MSD)

Example 1-29

Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on PO, PI, and P2.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char x, binbyte, d1, d2, d3;
    binbyte = 0xFD;         //binary(hex) byte
    x = binbyte / 10;       //divide by 10
    d1 = binbyte % 10;      //find remainder (LSD)
    d2 = x % 10;            //middle digit
    d3 = x / 10;            //most significant digit (MSD)
    P0 = d1;
    P1 = d2;
    P2 = d3;
  }
```

3.5 Accessing code ROM space in 8051C

Using the code (program) space for predefined data is the widely used option in the 8051.

RAM data space v. code data space

In the 8051 we have three spaces in which to store data. They are as follows:

1. The 128 bytes of RAM space with address range 00 – 7FH. (In the 8052, it is 256 bytes.) We can read (from) or write (into) this RAM space directly or indirectly using the RO and Rl registers.
2. The 64K bytes of code (program) space with addresses of 0000 – FFFFH. This 64K bytes of on-chip ROM space is used for storing programs (opcodes) and therefore is directly under the control of the program counter (PC).
3. There are two problems with using this code space for data.

   a) First, since it is ROM memory, we can burn our predefined data and tables into it. But we cannot write into it during the execution of the program.

   b) The second problem is that the more of this code space we use for data, the less is left for our program code. For example, if we have an 8051 chip such as DS89C420 with only 16K bytes of on-chip ROM, and we use 4K bytes of it to store some look-up table, only 12K bytes is left for the code program. For some applications this can be a problem. For this reason Intel created another memory space called *external memory* especially for data.
4. The 64K bytes of external memory, which can be used for both RAM and ROM. This 64K bytes is called external since we must use the MOVX Assembly language instruction to access it. At the time the 8051 was designed, the cost of on-chip ROM was very high; therefore, Intel used all the on-chip ROM for code but allowed connection to external RAM and ROM. In other words, we have a total of 128K bytes of memory space since the off-chip or external memory space of 64K bytes plus the 64K bytes of on-chip space provides you a total of 128K bytes of memory space.

3.5 .1 RAM data space usage by the 8051 C compiler

In Assembly language programming, as shown in Chapters 2 and 5, the 128 bytes of RAM space is used mainly by register banks and the stack. Whatever remains is used for scratch pad RAM. The 8051 C compiler first allocates the first 8 bytes of the RAM to bank 0 and then some RAM to the stack. Then it starts to allocate the rest to the variables declared by the C program. While in Assembly the default starting address for the stack is 08, the C compiler moves the stack's starting address to somewhere in the range of 50 – 7FH. This allows us to allocate contiguous RAM locations to array elements.

In cases where the program has individual variables in addition to array elements, the 8051 C compiler allocates RAM locations in the following order:

1. Bank 0 addresses 0-7
2. Individual variables addresses 08 and beyond
3. Array elements addresses right after variables
4. Stack addresses right after array elements
5. You can verify the above order by running Example 7-30 on your 8051 C simulator and examining the contents of the data RAM space. Remember that array elements need contiguous RAM locations and that limits the size of the array due to the fact that we have only 128 bytes of RAM for everything. In the case of Example 1-31 the array elements are limited to around 100.

Example 1-30

Compile and single-step the following program on your 8051 simulator. Examine the contents of the 128-byte RAM space to locate the ASCII values.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mynum[]= "ABCDEF";  //This uses RAM space
                                      //to store data
    unsigned char z;
    for(z=0;z<=6;z++)
      P1=mynum [z];
  }
```

Example 1-31

Write, compile, and single-step the following program on your 8051 simulator. Examine the contents of the code space to locate the values.

Solution:

```
#include <reg51.h>
void main(void)
  {
    unsigned char mydata[100];  //100 byte space in RAM
    unsigned char x,z=0;
    for(x=0;x<100;x++)
      {
        z--;                    //count down
        mydata[x]=z;            //save it in RAM
        P1=z;                   //give a copy to P1 too
      }
  }
```

3.5.2 Accessing code data space in 8051 C

To make the C compiler use the code space instead of the RAM space, we need to put the keyword *code* in front of the variable declaration. The following are some examples:

```
code unsigned char mynum[]= "012345ABCD";  //use code space
code unsigned char weekdays=7, month=0x12; //use code space
```

Example 1-32

Compile and single-step the following program on your 8051 simulator. Examine the contents of the code space to locate the ASCII values.

Solution:

```
#include <reg51.h>
void main(void)
  {
    code unsigned char mynum[]= "ABCDEF";  //uses code space
                                           //for data
    unsigned char z;
    for(z=0;z<=6;z++)
      P1=mynum[z];
  }
```

Compiler variations

Example 1-33. It shows three different versions of a program that sends me string “HELLO” to the PI port. Compile each program with the 8051 C compiler of your choice and compare

the hex file size. Then compile each program on a different 8051 C compiler, and examine the hex file size to see the effectiveness of your C compiler.

Example 1-33

Compare and contrast the following programs and discuss the advantages and disadvantages of each one.

(a)

```
#include <reg51.h>
void main(void)
  {
    P1='H';
    P1='E';
    P1='L';
    P1='L';
    P1='O';
  }
```

(b)

```
#include <reg51.h>
void main(void)
  {
    unsigned char mydata[]="HELLO";
    unsigned char z;
    for(z=0;z<=5;z++)
      P1=mydata[z];
  }
```

(c)

```
#include <reg51.h>
void main(void)
  {
    //Notice Keyword code
    code unsigned char mydata[]="HELLO";
    unsigned char z;
    for(z=0;z<=5;z++)
      P1=mydata[z];
  }
```

Solution:

All the programs send out “HELLO” to PI, one character at a time, but they do it in different ways. The first one is short and simple, but the individual characters are embedded into the program. If we change the characters, the whole program changes. It also mixes the code and data together. The second one uses the RAM data space to store array elements, therefore the size of the array is limited. The third one uses a separate area of the code space for data. This allows the size of the array to be as long as you want if you have the on-chip ROM. However, the more code space you use for data, the less space is left for your program code. Both programs (b) and (c) are easily upgradable if we want to change the string itself or make it longer. That is not the case for program (a).

3.6 Data serialization using 8051C

Serializing data is a way of sending a byte of data one bit at a time through a single pin of microcontroller. There are two ways to transfer a byte of data serially:

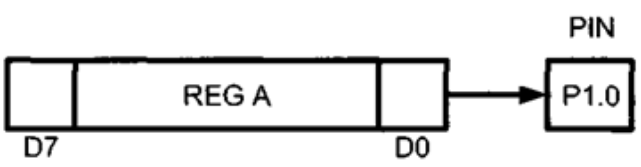
1. Using the serial port. When using the serial port, the programmer has very limited control over the sequence of data transfer. The detail of serial port data transfer is discussed in Chapter 10.
2. The second method of serializing data is to transfer data one bit a time and control the sequence of data and spaces in between them. In many new generations of devices such as LCD, ADC, and ROM the serial versions are becoming popular since they take less space on a printed circuit board.

Example 1-34

Write a C program to send out the value 44H serially one bit at a time via Pl.O. The LSB should go out first.

Solution:

```
//SERIALIZING DATA VIA P1.0 (SHIFTING RIGHT)
#include <reg51.h>
sbit  P1b0 = P1^0;
sbit  regALSB = ACC^0;
void main(void)
  {
    unsigned char conbyte = 0x44;
    unsigned char x;
    ACC = conbyte;
    for(x=0; x<8; x++)
      {
        P1b0 = regALSB;
        ACC = ACC >> 1;
      }
  }
```



Example 1-35

Write a C program to send out the value 44H serially one bit at a time via Pl.O. The MSB should go out first.

Solution:

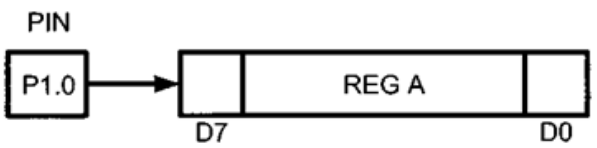
```
//SERIALIZING DATA VIA P1.0 (SHIFTING LEFT)
#include <reg51.h>
sbit P1b0 =  P1^0;
sbit regAMSB = ACC^7;
void main(void)
  {
    unsigned char conbyte = 0x44;
    unsigned char x;
    ACC = conbyte;
    for(x=0; x<8; x++)
      {
        P1b0 = regAMSB;
        ACC = ACC << 1;
      }
  }
```

Example 1-36

Write a C program to bring in a byte of data serially one bit at a time via PI .0. The LSB should come in first.

Solution:

```
//BRINGING IN DATA VIA P1.0 (SHIFTING RIGHT)
#include <reg51.h>
sbit P1b0 =  P1^0;
sbit ACCMSB = ACC^7;
void main(void)
  {
    unsigned char conbyte = 0x44;
    unsigned char x;
    for(x=0; x<8; x++)
      {
        ACCMSB = P1b0;
        ACC = ACC >> 1;
      }
    P2=ACC;
  }
```



3.7 Programming 8051 timers

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

1. The 16-bit register of Timer 0 is accessed as low byte and high byte. The low byte register is called TLO (Timer 0 low byte) and the high byte register is referred to as THO (Timer 0 high byte).
2. These registers can be accessed like any other register, such as A, B, RO, Rl, R2, etc. For example, the instruction "MOV TLO , #4FH" moves the value 4FH into TLO, the low byte of Timer 0.
3. These registers can also be read like any other register. For example, "MOV R5 , THO" saves THO (high byte of Timer 0) in R5.

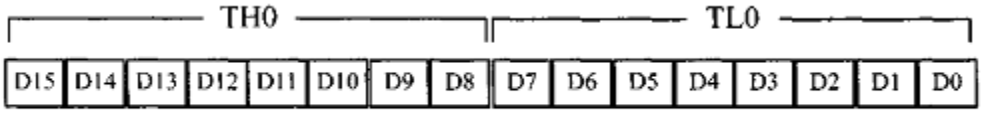


Figure 3.1.: Timer 0 Registers

Timer 1 registers

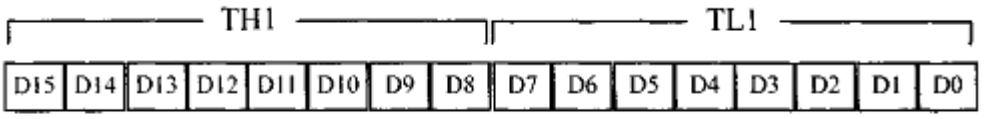
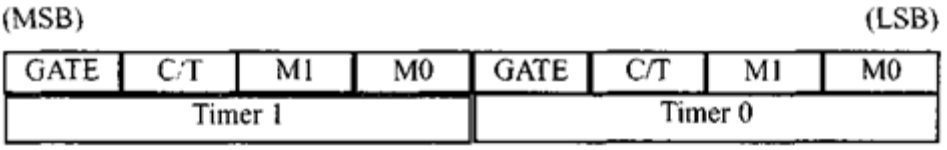


Figure 3.2.: Timer 0 Registers

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.

3.7.1 TMOD (timer mode) register

1. 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 set aside for Timer 0 and the upper 4 bits for Timer 1.
2. In each case, the lower 2 bits are used to set the timer mode and the upper 2 bits to specify the operation.



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

Figure 3.3:TMOD Register

M1, MO

MO and Ml select the timer mode. As shown in Figure 9-3, 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. We will concentrate on modes 1 and 2 since they are the ones used most widely. We will soon describe the characteristics of these modes, after describing the rest of the TMOD register.

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 1-37

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

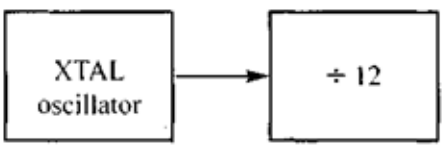
As you know, every timer needs a clock pulse to tick. What is the source of the clock pulse for the 8051 timers? If C T = 0. the crystal frequency attached to the 8051 is the source of the clock for the timer. This means that the size of the crystal frequency attached to the 8051 also decides the speed 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 1-38

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

NOTE THAT 8051 TIMERS USE 1/12 OF XTAL FREQUENCY, REGARDLESS OF MACHINE CYCLE TIME.

Example 1-39

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.

Now that we have this basic understanding of the role of the TMOD register, we will look at the timer's modes and how they are programmed to create a time delay. Because modes 1 and 2 are so widely used, we describe each of them in detail.

Although various 8051-based systems have an XTAL frequency of 10 MHz to 40 MHz, we will concentrate on the XTAL frequency of 11.0592 MHz. The reason behind such an odd number has to do with the baud rate for serial communication of the 8051. XTAL = 11.0592 MHz allows the 8051 system to communicate with the IBM PC with no errors.

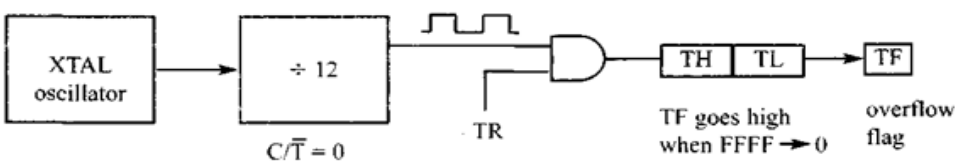
GATE

1. The other bit of the TMOD register is the GATE bit. Notice in the TMOD register of that both Timers 0 and 1 have the GATE bit.
2. 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 timers in the 8051 have both.
3. The start and stop of the timer are controlled by way of software by the TR (timer start) bits TRO and TR1. This is achieved by the instructions "SETB TR1″ and "CLR TR1″ for Timer 1, and "SETB TRO" and "CLR TRO" for Timer 0.
4. 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.
5. The hardware way of starting and stopping the timer by an external source is achieved by making GATE = 1 in the TMOD register.
6. However, to avoid further confusion for now, we will make GATE = 0, meaning that no external hardware is needed to start and stop the timers. In using software to start and stop the timer where GATE = 0. all we need are the instructions "SETB TRx" and "CLR TRx".

3.7.2 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 start ed. This is done by "SETB TRO" 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 flag bit called TF (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 TRO" or "CLR TR1″, for Timer 0 and Timer 1, respectively. Again, it must be noted that each timer has its own timer flag: TFO for Timer 0, and TF1 for Timer 1.
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. To clarify these steps, see Example 1-40

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.
1. Load registers TL and TH with initial count values.
2. Start the timer.
    1. 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.
3. Stop the timer.
4. Clear the TF flag for the next round.
5. Go back to Step 2 to load TH and TL again.

To calculate the exact time delay and the square wave frequency generated on pin P1 .5, we need to know the XTAL frequency.

The scientific calculator in the Accessories directory of Microsoft Windows can help you to find the TH, TL values. This calculator supports decimal, hex, and binary calculations.

(a) in hex

(FFFF – YYXX + 1) X 1.085 us where YYXX are TH, TL initial values respectively. Notice that values YYXX are in hex.

(b) in decimal

Convert YYXX values of the TH,TL register to decimal to get a NNNNN decimal number, then (65536 – NNNNN) x 1.085 mircosec

Timer Delay Calculation for XTAL = 11.0592 MHz

Example 1-40

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
```

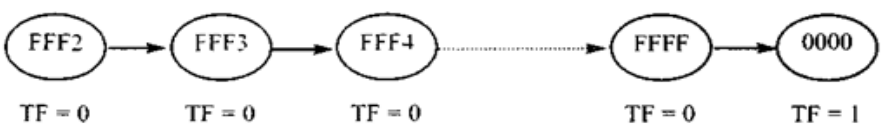
Solution:

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.

1. 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.
2. Timer 0 is stopped by the instruction "CLR TRO". The DELAY subroutine ends,
   and the process is repeated.

Notice that to repeat the process, we must reload the TL and TH registers and start the timer again.



Example 1-41

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. In other words, 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 1-42

Calculate the frequency of the square wave generated on pin P1. 5.

Solution:

In the time delay calculation of Example 9-5, 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. To do that, we use the machine cycles from Table A-l in Appendix A, as shown below.

```
                                              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
```

$T = 2 \times 28 \times 1.085\ \mu s = 60.76\ \mu s$ and $F = 16458.2$ Hz.

NOTE THAT 8051 TIMERS USE 1/12 OF XTAL FREQUENCY, REGARDLESS OF MACHINE CYCLE TIME.

Example 1-43

Find the delay generated by Timer 0 in the following code, using both of the methods of Figure 3.3. 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 µs= 19.93145ms.
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 1.085µs in duration. Therefore, we have 18370 x 1.085 µs = 19.93145 ms as the width of the pulse.

Example 1-44

Modify TL and TH in Example 9-7 to get the largest time delay possible. Find the delay in ms. In your calculation, exclude the overhead due to the instructions in the loop.

Solution:

To get the largest delay we make TL and TH both 0. This will count up from 0000 to FFFFH and then roll over to zero.

```
        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,#0        ;TH0 = 0, High byte
        SETB  P2.3          ;SET P2.3 high
        SETB  TR0           ;start Timer 0
AGAIN:  JNB   TF0,AGAIN     ;monitor Timer 0 flag
        CLR   TR0           ;stop Timer 0
        CLR   TF0           ;clear Timer 0 flag
        CLR   P2.3
```

Making TH and TL both zero means that the timer will count from 0000 to FFFFH, and then roll over to raise the TF flag. As a result, it goes through a total of 65536 states. Therefore, we have delay = (65536 – 0) x 1.085 µs = 71.1065 ms.

Example 1-45

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

1. Assuming that we know the amount of timer delay we need, the question is how to find the values needed for the TH, TL registers. To calculate the values to be loaded into the TL and TH registers look at Example 9-10 where we use crystal frequency of 11.0592 MHz for the 8051 system.
2. Assuming XTAL = 11.0592 MHz from Example 9-10 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.
    1. Convert the result of Step 2 to hex, where *yyxx* is the initial hex value to be loaded into the timer's registers.
3. Set TL = *xx* and TH = *yy*.

Example 1-46

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.

Solution:

Since XTAL = 11.0592 MHz, the counter counts up every 1.085 us. This means that out of many 1.085 us intervals we must make a 5 ms pulse. To get that, we divide one by the other. We need 5 ms /•! .085 us = 4608 clocks. To achieve that we need to load into TL and TH the value 65536 – 4608 = 60928 = EEOOH. Therefore, we have TH = EE and tt = on

```
        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 1-47

Assuming that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 kHz frequency on pin P1 .5.

Solution:

Look at the following steps.

1. T = 1 / f = 1 / 2 kHz = 500 µs the period of the square wave.
2. 1/2 of it for the high and low portions of the pulse is 250 µs.
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
```

Example 1-48

Assuming XTAL = 11.0592 MHz, write a program to generate a square wave of 50 Hz frequency on pin P2.3.

Solution:

Look at the following steps.

1. T — 1 / 50 Hz = 20 ms, the period of the square wave.
2. 1/2 of it for the high and low portions of the pulse = 10 ms
3. 10 ms / 1.085 us = 9216 and 65536 – 9216 = 56320 in decimal, and in hex it is DCOOH.
4. TL = 00 and TH = DC (hex)

The program follows.

```
        MOV   TMOD,#10H      ;Timer 1, mode 1 (16-bit)
AGAIN:  MOV   TL1,#00        ;TL1 = 00, Low byte
        MOV   TH1,#0DCH      ;TH1 = DCH, High byte
        SETB  TR1            ;start Timer 1
BACK:   JNB   TF1,BACK       ;stay until timer rolls over
        CLR   TR1            ;stop Timer 1
        CPL   P2.3           ;comp. P2.3 to get hi, lo
        CLR   TF1            ;clear Timer 1 flag
        SJMP  AGAIN          ;reload timer since mode 1
                             ;is not auto-reload
```

3.7.3 Generating a large time delay

As we have seen in the examples so far, the size of the time delay depends on two factors,

(a) The crystal frequency, and

(b) The timer's 16-bit register in mode 1.

Both of these factors are beyond the control of the 8051 programmer. We saw earlier that the largest time delay is achieved by making both TH and TL zero.

Using Windows calculator to find TH, TL

The scientific calculator in Microsoft Windows is a handy and easy-to-use tool to find the TH, TL values. Assume that we would like to find the TH, TL values for a time delay that uses 35,000 clocks of 1.085 us. The following steps show the calculation.

1. Bring up the scientific calculator in MS Windows and select decimal.
2. Enter 35,000.

3. Select hex. This converts 35,000 to hex, which is 88B8H.
4. Select +/- to give -35000 decimal (7748H).
    1. The lowest two digits (48) of this hex value are for TL and the next two (77) are for TH. We ignore all the Fs on the left since our number is 16-bit data.

Example 1-49

Examine the following program and find the time delay in seconds. Exclude the overhead due to the instructions in the loop.

```
        MOV   TMOD,#10H      ;Timer 1, mode 1(16-bit)
        MOV   R3,#200        ;counter for multiple delay
AGAIN:  MOV   TL1,#08H       ;TL1 = 08, Low byte
        MOV   TH1,#01H       ;TH1 = 01, High byte
        SETB  TR1            ;start Timer 1
BACK:   JNB   TF1,BACK       ;stay until timer rolls over
        CLR   TR1            ;stop Timer 1
        CLR   TF1            ;clear Timer 1 flag
        DJNZ  R3,AGAIN       ;if R3 not zero then
                             ;reload timer
```

Solution:

TH – TL = 0108H = 264 in decimal and 65536 – 264 = 65272. Now 65272 x 1.085 μs = 70.820 ms, and for 200 of them we have 200 x 70.820 ms = 14.164024 seconds.

3.7.4 Mode 0

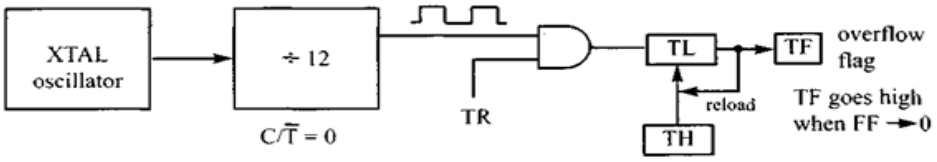
Mode 0 is exactly like mode 1 except that it is a 13-bit timer instead of 16-bit. The 13-bit counter can hold values between 0000 to 1FFFH in TH – TL. Therefore, when the timer reaches its maximum of 1FFH, it rolls over to 0000, and TF is raised.

3.7.5 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 register. 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 raised.



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.
5. 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. This reloading leaves TH unchanged, still holding a copy of the original value. This mode has many applications, including setting the baud rate in serial communication,

Steps to program in mode 2

To generate a time delay using the timer's mode 2, take the following steps.

1. Load the TMOD value register indicating which timer (Timer 0 or Timer 1) is to be used, and select the timer mode (mode 2).
2. Load the TH registers with the initial count value.
3. Start the timer.
4. 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.
5. Clear the TF flag.
6. Go back to Step 4, since mode 2 is auto-reload.

Example 1-50

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:

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 µs = 251 x 1.085 us = 272.33 µs 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.

1. To get the smallest frequency, we need the largest T and that is achieved when TH = 00.
2. In that case, we have T = 2 x 256 x 1.085 µs = 555.52 µs and the frequency = 1.8kHz.

Example 1-51

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.

Example 1-52

Assuming that we are programming the timers for mode 2, find the value (in hex) loaded into TH for each of the following cases.

```
(a) MOV    TH1,#-200      (b) MOV    TH0,#-60
(c) MOV    TH1,#-3        (d) MOV    TH1,#-12
(e) MOV    TH0,#-48
```

Solution:

You can use the Windows scientific calculator to verify the results provided by the assembler. In Windows calculator, select decimal and enter 200. Then select hex, then +/- to get the TH value. Remember that we only use the right two digits and ignore the rest since our data is an 8-bit data. The following is what we get.

Decimal	*2's complement (TH value)*
-200	38H
-60	C4H
-3	FDH
-12	F4H
-48	D0H

3.7.6 Assemblers and negative values

Since the timer is 8-bit in mode 2, we can let the assembler calculate the value for TH. For example, in "MOV TH1 ,#-10 0″, the assembler will calculate the -100 = 9C, and makes TH1 = 9C in hex. This makes our job easier.

Example 1-53

Find (a) the frequency of the square wave generated in the following code, and (b) the duty cycle of this wave.

```
          MOV   TMOD,#2H          ;Timer 0, mode 2
                                  ;(8-bit, auto-reload)
          MOV   TH0,#-150         ;TH0 = 6AH = 2's comp of -150
AGAIN:    SETB  P1.3              ;P1.3 = 1
          ACALL DELAY
          ACALL DELAY
          CLR   P1.3              ;P1.3 = 0
          ACALL DELAY
          SJMP  AGAIN
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
          RET
```

Solution:

For the TH value in mode 2, the conversion is done by the assembler as long as we enter a negative number. This also makes the calculation easy. Since we are using 150 clocks, we have time for the DELAY subroutine = 150 x 1.085 µs = 162 µs. The high portion of the pulse is twice that of the low portion (66% duty cycle). Therefore, we have: T = high portion + low portion = 325.5 (is + 162.25µs = 488.25 µs and frequency = 2.048 kHz.

Notice that in many of the time delay calculations we have ignored the clocks caused by the overhead instructions in the loop. To get a more accurate time delay, and hence frequency, you need to include them. If you use a digital scope and you don't get exactly the same frequency as the one we have calculated, it is because of the overhead associated with those instructions.

3.8 Counter programming

C/T bit in TMOD register

1. Recall from the last section that the C/T bit in the TMOD register decides the source of the clock for the timer. If C/T = 0, the timer gets pulses from the crystal.
2. In contrast, when C/T = 1, the timer is used as a counter and gets its pulses from outside the 8051. Therefore, when C/T = 1, the counter counts up as pulses are fed from pins 14 and 15.
3. These pins are called TO (Timer 0 input) and Tl (Timer 1 input). Notice that these two pins belong to port 3.
4. In the case of Timer 0, when C/T = *1*, pin P3.4 provides the clock pulse and the counter counts up for each clock pulse coming from that pin. Similarly, for Timer 1, when C/T = 1 each clock pulse coming in from pin P3.5 makes the counter count up.

Table 1.5 : Port 3 Pins Used For Timers 0 and 1

Pin	Port Pin	Function	Description
14	P3.4	T0	Timer/Counter 0 external input
15	P3.5	T1	Timer/Counter 1 external input

(MSB)							(LSB)
GATE	C/T	M1	M0	GATE	C/T	M1	M0
Timer 1				Timer 0			

Example 1-54

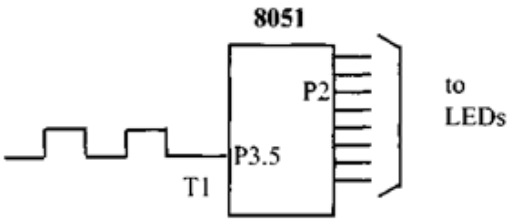
Assuming that clock pulses are fed into pin Tl, write a program for counter 1 in mode 2 to count the pulses and display the state of the TL1 count on **P2.**

Solution:

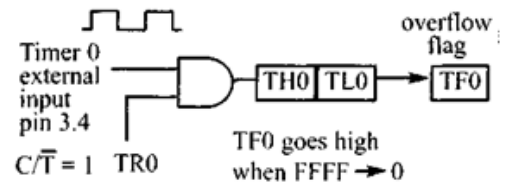
```
        MOV   TMOD,#01100000B     ;counter 1, mode 2,C/T=1
                                  ;external pulses
        MOV   TH1,#0              ;clear TH1
        SETB  P3.5                ;make T1 input
AGAIN:  SETB  TR1                 ;start the counter
BACK:   MOV   A,TL1               ;get copy of count TL1
        MOV   P2,A                ;display it on port 2
        JNB   TF1,BACK            ;keep doing it if TF=0
        CLR   TR1                 ;stop the counter 1
        CLR   TF1                 ;make TF=0
        SJMP  AGAIN               ;keep doing it
```

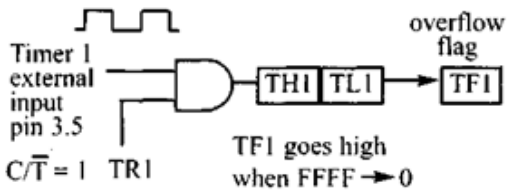
Notice in the above program the role of the instruction "SETB P3.5". Since ports are set up for output when the 8051 is powered up, we make P3.5 an input port by making it high. In other words, we must configure (set high) the T1 pin (pin P3.5) to allow pulses to be fed into it.

P2 is connected to 8 LEDs and input T1 to pulse.



In Example 1-54, we use Timer 1 as an event counter where it counts up as clock pulses are fed into pin 3.5. These clock pulses could represent the number of people passing through an entrance, or the number of wheel rotations, or any other event that can be converted to pulses. In Example 1-54, the TL data was displayed in binary. In Example 9-19, the TL registers are converted to ASCII to be displayed on an LCD.





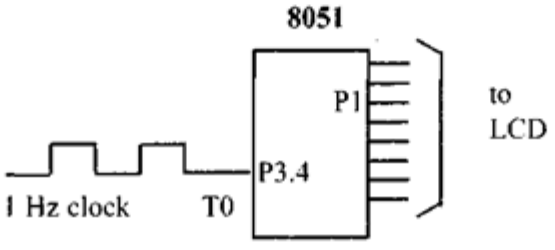
Example 1-55

Assume that a 1-Hz frequency pulse is connected to input pin 3.4. Write a program to display counter 0 on an LCD. Set the initial value of THO to -60.

Solution:

To display the TL count on an LCD, we must convert 8-bit binary data to ASCII. See Chapter 6 for data conversion.

```
            ACALL  LCD_SET_UP       ;initialize the LCD
            MOV    TMOD,#00000110B  ;counter 0,mode 2,C/T=1
            MOV    TH0,#-60         ;counting 60 pulses
            SETB   P3.4             ;make T0 as input
AGAIN:      SETB   TR0              ;starts the counter
BACK:       MOV    A,TL0            ;get copy of count TL0
            ACALL  CONV             ;convert in R2, R3, R4
            ACALL  DISPLAY          ;display on LCD
            JNB    TF0,BACK         ;loop if TF0=0
            CLR    TR0              ;stop the counter 0
            CLR    TF0              ;make TF0=0
            SJMP   AGAIN            ;keep doing it

;converting 8-bit binary to ASCII
;upon return, R4, R3, R2 have ASCII data (R2 has LSD)

CONV:       MOV   B,#10             ;divide by 10
            DIV   AB
            MOV   R2,B              ;save low digit
            MOV   B,#10             ;divide by 10 once more
            DIV   AB
            ORL   A,#30H            ;make it ASCII
            MOV   R4,A              ;save MSD
            MOV   A,B
            ORL   A,#30H            ;make 2nd digit an ASCII
            MOV   R3,A              ;save it
            MOV   A,R2
            ORL   A,#30H            ;make 3rd digit an ASCII
            MOV   R2,A              ;save the ASCII
            RET
```



By using 60 Hz we can generate seconds, minutes, hours.

Note that on the first round, it starts from 0, since on RESET, TLO = 0. To solve this problem, load TLO with -60 at the beginning of the program.

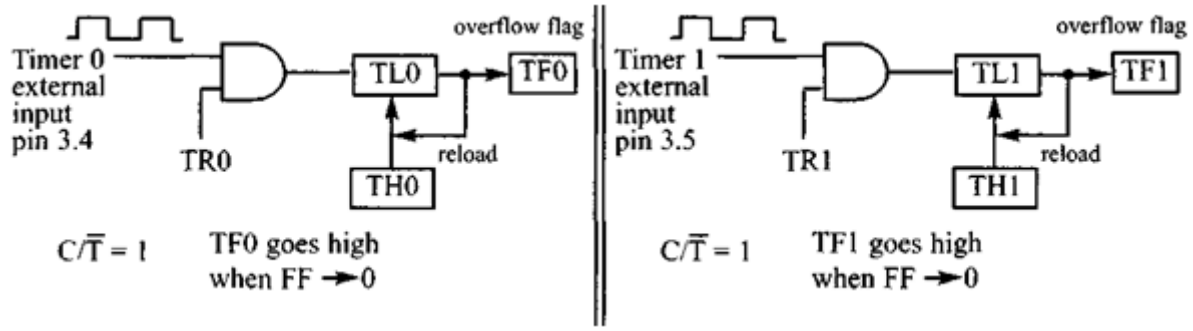


Figure 3.3 :Timer 0 with External Input (Mode 2) Figure 3.4 Timer 1 with External Input (Mode 2)

As another example of the application of the timer with C/T = 1, we can feed an external square wave of 60 Hz frequency into the timer. The program will generate the second, the minute, and the hour out of this input frequency and display the result on an LCD. This will be a nice digital clock, but not a very accurate one.

Table 1.5 : Equivalent Instructions for the Timer Control Register (TCON)

For Timer 0			
	SETB TR0	=	SETB TCON.4
	CLR TR0	=	CLR TCON.4
	SETB TF0	=	SETB TCON.5
	CLR TF0	=	CLR TCON.5
For Timer 1			
	SETB TR1	=	SETB TCON.6
	CLR TR1	=	CLR TCON.6
	SETB TF1	=	SETB TCON.7
	CLR TF1	=	CLR TCON.7

TCON: Timer/Counter Control Register

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

3.8.1 TCON register

1. In the examples so far we have seen the use of the TRO and TR1 flags to turn on or off the timers. These bits are part of a register called TCON (timer control). This register is an 8-bit register.
2. As shown in Table 1.5, the upper four bits are used to store the TF and TR bits of both Timer 0 and Timer 1. The lower four bits are set aside for controlling the interrupt bits. We must notice that the TCON register is a bit-addressable register. Instead of using instructions such as "SETB TR1″ and "CLR TR1″, we could use "SETB TCON. 6″ and "CLR TCON. 6″, respectively.

The case of GATE = 1 in TMOD

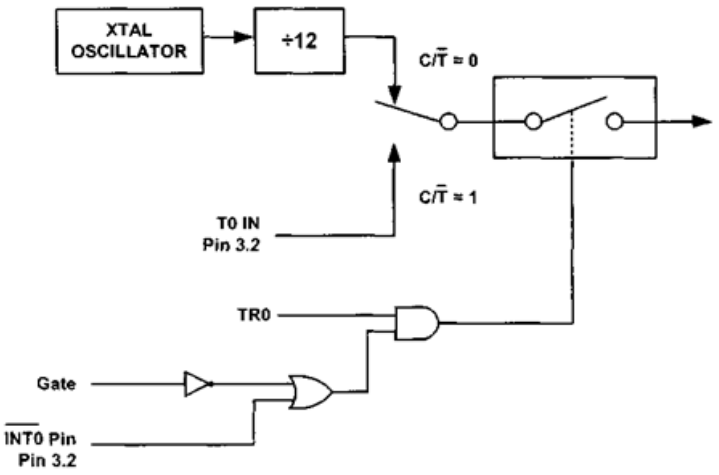


Figure 3.5. Timer/Counter 0

1. All discussion so far has assumed that GATE = 0. When GATE = 0, the timer is started with instructions "SETB TRO" and "SETB TR1″, for Timers 0 and 1, respectively.
2. What happens if the GATE bit in TMOD is set to 1.if GATE = 1, the start and stop of the timer are done externally through pins P3.2 and P3.3 for Timers 0 and 1, respectively.
3. This is in spite of the fact that TRx is turned on by the "SETB TRx" instruction. This allows us to start or stop the timer externally at any time via a simple switch. This hardware way of controlling the stop and start of the timer can have many applications.
4. For example, assume that an 8051 system is used in a product to sound an alarm every second using Timer 0, perhaps in addition to many other things. Timer 0 is turned on by the software method of using the "SETB TRO" instruction and is beyond

the control of the user of that product. However, a switch connected to pin P3.2 can be used to turn on and off the timer, thereby shutting down the alarm.

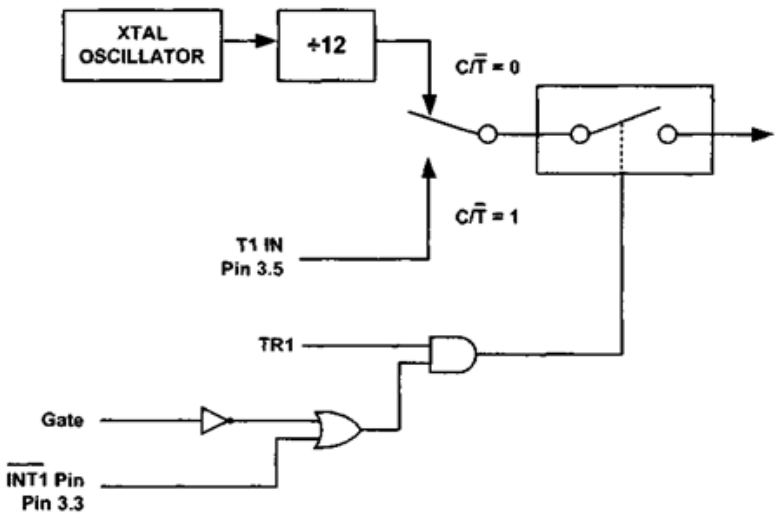


Figure 3.6. Timer/Counter 1

3.9 Programming timers 0 and 1 in 8051 C

Accessing timer registers in C

In 8051 C we can access the timer registers TH, TL, and TMOD directly using the reg51 .h header file.

Example 1-56

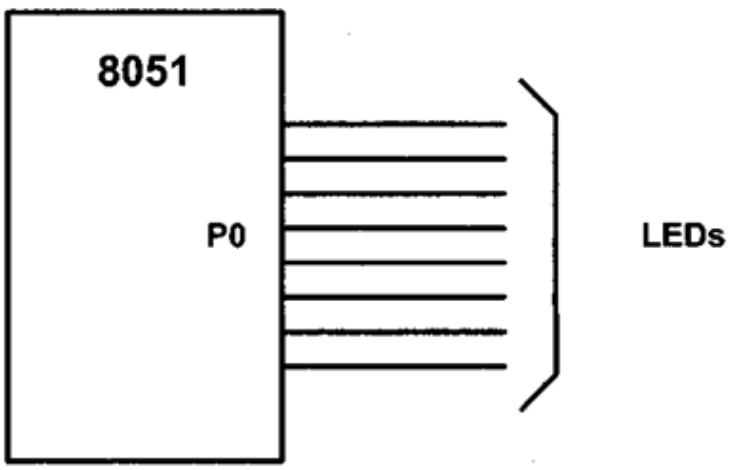
Write a 8051 C program to toggle all the bits of port P1 continuously with some delay in between. Use Timer 0, 16-bit mode to generate the delay.

Solution:

```
#include <reg51.h>
void T0Delay(void);
void main(void)
  {
   while(1)              //repeat forever
    {
      P1=0x55;           //toggle all bits of P1
      T0Delay();         //delay size unknown
      P1=0xAA;           //toggle all bits of P1
      T0Delay();
    }
  }

void T0Delay()
  {
    TMOD=0x01;           //Timer 0, Mode 1
    TL0=0x00;            //load TL0
    TH0=0x35;            //load TH0
    TR0=1;               //turn on T0
    while(TF0==0);       //wait for TF0 to roll over
    TR0=0;               //turn off T0
    TF0=0;               //clear TF0
  }
```

FFFFH – 3500H = CAFFH = 51967 + 1 = 51968

51968 × 1.085 μs = 56.384 ms is the approximate delay.



Calculating delay length using timers

Delay duration for the AT89C51/52 and DS89C4xO chips

As we stated before, there is a major difference between the AT89C51 and DS89C4xO chips in term of the time it takes to execute a single instruction. Although the DS89C4xO executes instructions 12 times faster than the AT89C51 chip, they both still use Osc/12 clock for their timers. The faster execution time for the instructions will have an impact on your delay length.

To verify this very important point, compare parts (a) and (b) of Example 9-21 since they have been tested on these two chips with the same speed and C compiler.

Timers 0 and 1 delay using mode 1 (16-bit non auto-reload)

Examples 9-21 and 9-22 show 8051 C programming of the timers 0 and 1 in mode 1 (16-bit non-auto reload). Examine them to get familiar with the syntax.

Timers 0 and 1 delay using mode 2 (8-bit auto-reload)

Study these examples below to get familiar with the syntax.

Example 1-57

Write an 8051 C program to toggle only bit PI.5 continuously every 50 ms. Use Timer 0, mode 1 (16-bit) to create the delay. Test the program (a) on the AT89C51 and (b) on the DS89C420.

Solution:

```
#include <reg51.h>
void T0M1Delay(void);
sbit mybit=P1^5;
void main(void)
   {
    while(1)
      {
        mybit=~mybit;   //toggle P1.5
        T0M1Delay();    //Timer 0, mode 1(16-bit)
      }
   }
```

(a) Tested for AT89C51, XTAL=11.0592 MHz, using the Proview32 compiler

```
void T0M1Delay(void)
   {
     TMOD=0x01;         //Timer 0, mode 1(16-bit)
     TL0=0xFD;          //load TL0
     TH0=0x4B;          //load TH0
     TR0=1;             //turn on T0
     while(TF0==0);     //wait for TF0 to roll over
     TR0=0;             //turn off T0
     TF0=0;             //clear TF0
   }
```

(b) Tested for DS89C420, XTAL=11.0592 MHz, using the Proview32 compiler

```
void T0M1Delay(void)
   {
     TMOD=0x01;         //Timer 0, mode 1(16-bit)
     TL0=0xFD;          //load TL0
     TH0=0x4B;          //load TH0
     TR0=1;             //turn on T0
     while(TF0==0);     //wait for TF0 to roll over
     TR0=0;             //turn off T0
     TF0=0;             //clear TF0
   }
```

FFFFH – 4BFDH = B402H = 46082 + 1 = 46083

Timer delay = 46083 × 1.085 μs = 50 ms

Example 1-58

Write an 8051 C program to toggle all bits of *P2* continuously every 500 ms. Use Timer 1. mode 1 to create the delay.

Solution:

```
//tested for DS89C420, XTAL = 11.0592 MHz, using the Proview32 compiler

#include <reg51.h>
void T1M1Delay(void);
void main(void)
  {
    unsigned char x;
    P2=0x55;
    while(1)
      {
        P2=~P2;       //toggle all bits of P2
        for(x=0;x<20;x++)
          T1M1Delay();
      }
  }

void T1M1Delay(void)
  {
    TMOD=0x10;        //Timer 1, mode 1(16-bit)
    TL1=0xFE;         //load TL1
    TH1=0xA5;         //load TH1
    TR1=1;            //turn on T1
    while(TF1==0);    //wait for TF1 to roll over
    TR1=0;            //turn off T1
    TF1=0;            //clear TF1
  }
```

A5FEH = 42494 in decimal

65536 – 42494 = 23042

23042 × 1.085 μs = 25 ms and 20 × 25 ms = 500 ms

Example 1-59

Write an 8051 C program to toggle only pin PI.5 continuously every 250 ms. Use Timer 0, mode 2 (8-bit auto-reload) to create the delay.

Solution:

//tested for DS89C420, XTAL = 11.0592 MHz, using the Proview32 compiler

```
#include <reg51.h>
void T0M2Delay(void);
sbit mybit=P1^5;
void main(void)
  {
    unsigned char x, y;
    while(1)
      {
        mybit=~mybit;            //toggle P1.5
        for(x=0;x<250;x++)       //due to for loop overhead
          for(y=0;y<36;y++)      //we put 36 and not 40
            T0M2Delay();
      }
  }

void T0M2Delay(void)
  {
    TMOD=0x02;               //Timer 0, mode 2(8-bit auto-reload)
    TH0=-23;                 //load TH0(auto-reload value)
    TR0=1;                   //turn on T0
    while(TF0==0);           //wait for TF0 to roll over
    TR0=0;                   //turn off T0
    TF0=0;                   //clear TF0
  }
```

256-23 = 233

23 x 1.085 us = 25 us

25 us x 250 x 40 = 250 ms by calculation.

However, the scope output does not give us this result. This is due to overhead of the for loop in C. To correct this problem, we put 36 instead of 40.

Example 1-60

Write an 8051 C program to create a frequency of 2500 Hz on pin P2.7. Use Timer 1. mode 2 to create the delay.

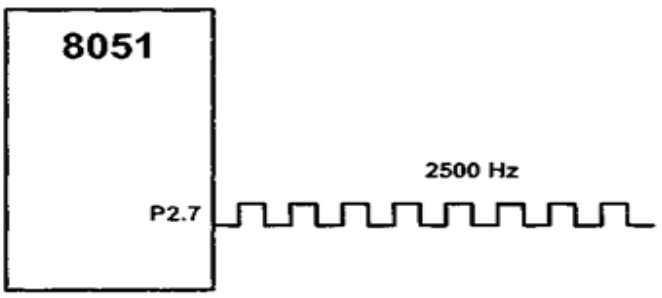
Solution:

```
//tested for DS89C420, XTAL = 11.0592 MHz, using the Proview32 compiler

#include <reg51.h>
void T1M2Delay(void);
sbit mybit=P2^7;
void main(void)
  {
    unsigned char x;
    while(1)
      {
        mybit=~mybit;       //toggle P2.7
          T1M2Delay();
      }
  }

void T1M2Delay(void)
  {
    TMOD=0x20;              //Timer 1, mode 2(8-bit auto-reload)
    TH1=-184;               //load TH1(auto-reload value)
    TR1=1;                  //turn on T1
    while(TF1==0);          //wait for TF1 to roll over
    TR1=0;                  //turn off T1
    TF1=0;                  //clear TF1
  }
```

1 / 2500 Hz = 400 μs

400 μs / 2 = 200 μs

200 μs / 1.085 μs = 184



Example 1-61

A switch is connected to pin PI.2. Write an 8051 C program to monitor SW and create the following frequencies on pin PI.7:

SW=0: 500 Hz

SW=1: 750 Hz

Use Timer 0, mode 1 for both of them.

Solution:

```
//tested for AT89C51/52, XTAL = 11.0592 MHz, using the Proview32 compiler
#include <reg51.h>
sbit mybit=P1^5;
sbit SW=P1^7;
void T0M1Delay(unsiged char);
void main(void)
  {
    SW=1;                    //make P1.7 an input
    while(1)
      {
        mybit=~mybit;        //toggle P1.5
        if(SW==0)            //check switch
          T0M1Delay(0);
        else
          T0M1Delay(1);
      }
  }
void T0M1Delay(unsigned char c)
  {
    TMOD=0x01;
    if(c==0)
      {
        TL0=0x67;            //FC67
        TH0=0xFC;
      }
    else
      {
        TL0=0x9A;            //FD9A
        TH0=0xFD;
      }
    TR0=1;
    while(TF0==0);
    TR0=0;
    TF0=0;
  }
```

FC67H = 64615
65536 – 64615 = 921
921 × 1.085 µs = 999.285 µs
1 / (999.285 µs × 2) = 500 Hz

3.9.1 C Programming of timers 0 and 1 as counters

A timer can be used as a counter if we provide pulses from outside the chip instead of using the frequency of the crystal oscillator as the clock source. By feeding pulses to the TO (P3.4) and Tl (P3.5) pins, we turn Timer 0 and Timer 1 into counter 0 and counter 1, respectively. Study the next few examples to see how timers 0 and 1 are programmed as counters using the C language.

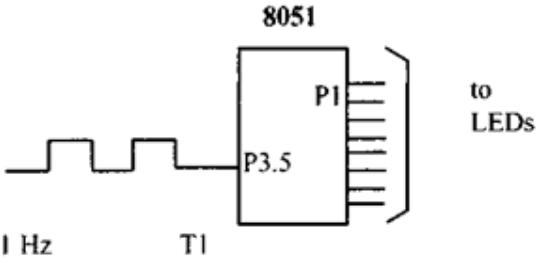
Example 1-62

Assume that a 1-Hz external clock is being fed into pin Tl (P3.5). Write a C program for counter 1 in mode 2 (8-bit auto reload) to count up and display the state of the TL1 count on PI. Start the count at OH.

Solution:

```
#include <reg51.h>
sbit T1 = P3^5;
void main(void)
  {
    T1=1;                   //make T1 an input
    TMOD=0x60;              //
    TH1=0;                  //set count to 0

    while(1)                //repeat forever
      {
       do
        {
          TR1=1;            //start timer
          P1=TL1;           //place value on pins
        }
       while(TF1==0);       //wait here
       TR1=0;               //stop timer
       TF1=0;               //clear flag
     }
  }
```

P1 is connected to 8 LEDs.
T1 (P3.5) is connected to a
1-Hz external clock.

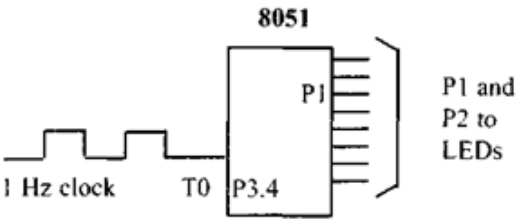


Example 1-63

Assume that a 1-Hz external clock is being fed into pin TO (P3.4). Write a C program for counter 0 in mode -1 (16-bit) to count the pulses and display the THO and TLO registers on P2 and PI, respectively.

Solution:

```
#include <reg51.h>

void main(void)
  {
    T0=1;                    //make T0 an input
    TMOD=0x05;               //
    TL0=0;                   //set count to 0
    TH0=0;                   //set count to 0

    while(1)                 //repeat forever
      {
       do
        {
          TR0=1;             //start timer
          P1=TL0;            //place value on pins
          P2=TH0;            //
        }
       while(TF0==0);        //wait here
       TR0=0;                //stop timer
       TF0=0;
      }
  }
```



Example 1-64

Assume that a 2-Hz external clock is being fed into pin Tl (P3.5). Write a C **program** for counter 0 in mode 2 (8-bit auto reload) to display the count in ASCII. The 8-bit binary count must be converted to ASCII. Display the ASCII digits (in binary) on PO, PI, and P2 where PO has the least significant digit. Set the initial value of THO to 0.

Solution:

To display the TL1 count we must convert 8-bit binary data to ASCII. See Chapter 7 for data conversion. The ASCII values will be shown in binary. For example, '9′ will show as 00111001 on ports.

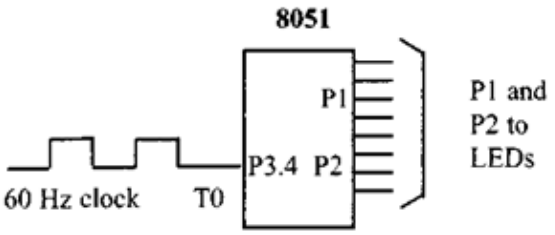
```
#include <reg51.h>
void BinToASCII(unsigned char);
void main()
  {
    unsigned char value;
    T1=1;
    TMOD=0x06;
    TH0=0;

    while(1)
      {
       do
        {
          TR0=1;
          value=TL0;
          BinToASCII(value);
        }
       while(TF0==0);
       TR0=0;
       TF0=0;
      }
  }

void BinToASCII(unsigned char value)        //see Chapter 7
  {
    unsigned char x,d1,d2,d3;
    x = value / 10;
    d1 = value % 10
    d2 = x % 10;
    d3 = x / 10
    P0 = 30 | d1;
    P1 = 30 | d2;
    P2 = 30 | d3
  }
```

Example 1-65

Assume that a 60-Hz external clock is being fed into pin TO (P3.4). Write a C program for counter 0 in mode 2 (8-bit auto-reload) to display the seconds and minutes on PI and P2, respectively.

Solution:

```
#include <reg51.h>
void ToTime(unsigned char);
void main()
  {
    unsigned char val;
    T0=1;
    TMOD=0x06;              //T0, mode 2, counter
    TH0=-60;                //sec = 60 pulses
    while(1)
      {
       do
        {
          TR0=1;
          sec=TL0;
          ToTime(val);
        }
       while(TF0==0);
       TR0=0;
       TF0=0;
      }
  }

void ToTime(unsigned char val)
  {
    unsigned char sec, min;
    min = value / 60;
    sec = value % 60;
    P1 = sec;
    P2 = min;
  }
```



By using 60 Hz, we can generate seconds, minutes, hours.

Outcomes

At the end of the module, students will be able to:

Evaluate software delays, timer delays and timer programming using both Assembly and C language.

[L5, MODULE 3]