Pearson New International Edition

The 8051 Microcontroller and Embedded Systems

Mazidi Mazidi McKinlay

Second Edition

Pearson New International Edition

The 8051 Microcontroller and
Embedded Systems
Mazidi   Mazidi   McKinlay
Second Edition

PEARSON

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England and Associated Companies throughout the world

Visit us on the World Wide Web at: www.pearsoned.co.uk

PEARSON

ISBN 10: 1-292-02657-X
ISBN 13: 978-1-292-02657-2

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Printed in the United States of America

... man's glory lieth in his knowledge,
his upright conduct, his praiseworthy character,
his wisdom, and not in his nationality or rank.

– Baha'u'llah

CONTENTS AT A GLANCE

CONTENTS

CHAPTER 4

I/O PORT PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> List the 4 ports of the 8051**
- **>> Describe the dual role of port 0 in providing both data and addresses**
- **>> Code Assembly language to use the ports for input or output**
- **>> Explain the dual role of port 0 and port 2**
- **>> Code 8051 instructions for I/O handling**
- **>> Code I/O bit-manipulation programs for the 8051**

This chapter describes the I/O port programming of the 8051 with many examples. In Section 4.1, we describe I/O access using byte-size data, and in Section 4.2, bit manipulation of the I/O ports is discussed in detail.

SECTION 4.1: 8051 I/O PROGRAMMING

In the 8051 there are a total of four ports for I/O operations. Examining Figure 4-1, note that of the 40 pins, a total of 32 pins are set aside for the four ports P0, P1, P2, and P3, where each port takes 8 pins. The rest of the pins are designated as V_{CC}, GND, XTAL1, XTAL2, RST, $\overline{EA}$, ALE/$\overline{PROG}$ and $\overline{PSEN}$ are discussed in Chapter 8.

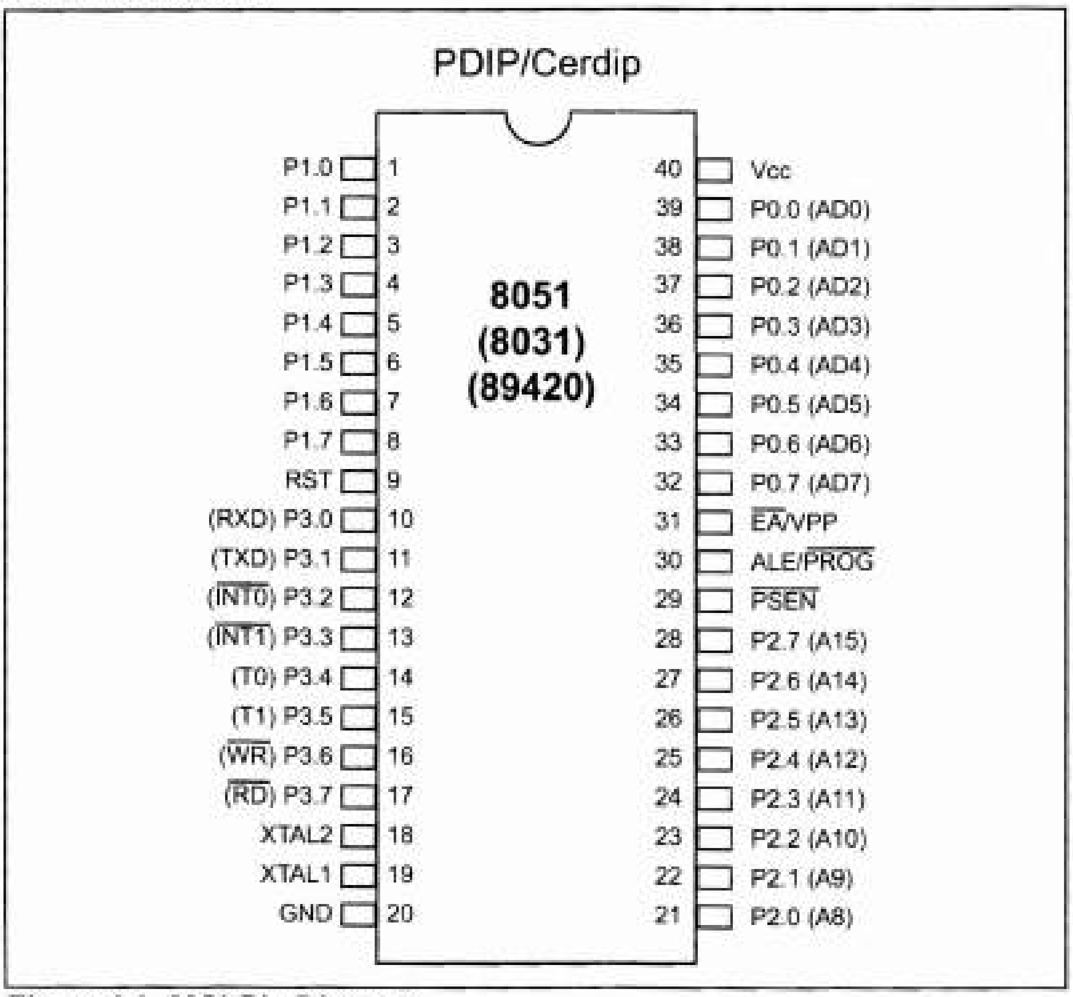


Figure 4-1. 8051 Pin Diagram

I/O port pins and their functions

The four ports P0, P1, P2, and P3 each use 8 pins, making them 8-bit ports. All the ports upon RESET are configured as inputs, ready to be used as input ports. When the first 0 is written to a port, it becomes an output. To reconfigure it as an input, a 1 must be sent to the port. To use any of these ports as an input port, it must be programmed, as we will explain throughout this section. First, we describe each port.

Port 0

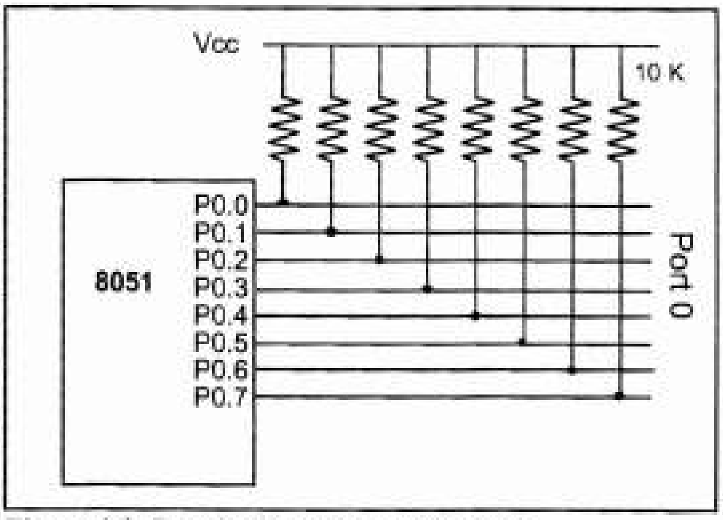


Figure 4-2. Port 0 with Pull-Up Resistors

Port 0 occupies a total of 8 pins (pins 32 - 39). It can be used for input or output. To use the pins of port 0 as both input and output ports, each pin must be connected externally to a 10K-ohm pull-up resistor. This is due to the fact that P0 is an open drain, unlike P1, P2, and P3, as we will soon see. *Open drain* is a term used for MOS chips in the same way that *open collector* is used for TTL chips. In any system using the 8051/52 chip, we normally connect P0 to pull-up resistors. See Figure 4-2. In this way we take advantage of port 0 for both input and output. For example, the following code will continuously send out to port 0 the alternating values of 55H and AAH.

```
;Toggle all bits of P0
BACK:     MOV   A,#55H
          MOV   P0,A
          ACALL DELAY
          MOV   A,#0AAH
          MOV   P0,A
          ACALL DELAY
          SJMP  BACK
```

It must be noted that complementing 55H (01010101) turns it into AAH (10101010). By sending 55H and AAH to a given port continuously, we toggle all the bits of that port.

Port 0 as input

With resistors connected to port 0, in order to make it an input, the port must be programmed by writing 1 to all the bits. In the following code, port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1.

```
;Get a byte from P0 and send it to P1
          MOV  A,#0FFH    ;A = FF hex
          MOV  P0,A       ;make P0 an input port
                          ;by writing all 1s to it
BACK:     MOV  A,P0       ;get data from P0
          MOV  P1,A       ;send it to port 1
          SJMP BACK       ;keep doing it
```

Dual role of port 0

As shown in Figure 4-1, port 0 is also designated as AD0 - AD7, allowing it to be used for both address and data. When connecting an 8051/31 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save pins. We discuss that in Chapter 14.

Port 1

Port 1 occupies a total of 8 pins (pins 1 through 8). It can be used as input or output. In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, port 1 is configured as an input port. The following code will continuously send out to port 1 the alternating values 55H and AAH.

```
;Toggle all bits of P1 continuously
          MOV    A,#55H
BACK:     MOV    P1,A
          ACALL  DELAY
          CPL    A           ;complement(Invert) reg. A
          SJMP   BACK
```

Port 1 as input

If port 1 has been configured as an output port, to make it an input port again, it must programmed as such by writing 1 to all its bits. The reason for this is discussed in Appendix C.2. In the following code, port 1 is configured first as an input port by writing 1s to it, then data is received from that port and saved in R7, R6, and R5.

```
          MOV    A,#0FFH   ;A=FF hex
          MOV    P1,A      ;make P1 an input port
                           ;by writing all 1s to it
          MOV    A,P1      ;get data from P1
          MOV    R7,A      ;save it in reg R7
          ACALL  DELAY     ;wait
          MOV    A,P1      ;get another data from P1
          MOV    R6,A      ;save it in reg R6
          ACALL  DELAY     ;wait
          MOV    A,P1      ;get another data from P1
          MOV    R5,A      ;save it in reg R5
```

Port 2

Port 2 occupies a total of 8 pins (pins 21 through 28). It can be used as input or output. Just like P1, port 2 does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, port 2 is configured as an input port. The following code will send out continuously to port 2 the alternating values 55H and AAH. That is, all the bits of P2 toggle continuously.

```
             MOV    A,#55H
BACK:        MOV    P2,A
             ACALL  DELAY
             CPL    A           ;complement reg. A
             SJMP   BACK
```

Port 2 as input

To make port 2 an input, it must programmed as such by writing 1 to all its bits. In the following code, port 2 is configured first as an input port by writing 1s to it. Then data is received from that port and is sent to P1 continuously.

```
;Get a byte from P2 and send it to P1
             MOV  A,#0FFH     ;A=FF hex
             MOV  P2,A        ;make P2 an input port by
                              ;writing all 1s to it
BACK:        MOV  A,P2        ;get data from P2
             MOV  P1,A        ;send it to Port 1
             SJMP BACK        ;keep doing that
```

Dual role of port 2

In many systems based on the 8051, P2 is used as simple I/O. However, in 8031-based systems, port 2 must be used along with P0 to provide the 16-bit address for external memory. As shown in Figure 4-1, port 2 is also designated as A8 - A15, indicating its dual function. Since an 8051/31 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0 - A7, it is the job of P2 to provide bits A8 - A15 of the address. In other words, when the 8051/31 is connected to external memory, P2 is used for the upper 8 bits of the 16-bit address, and it cannot be used for I/O. This is discussed in detail in Chapter 14.

From the discussion so far, we conclude that in systems based on 8751, 89C51, or DS589C4x0 microcontrollers, we have three ports, P0, P1, and P2, for I/O operations. This should be enough for most microcontroller applications. That leaves port 3 for interrupts as well as other signals, as we will see next.

Port 3

Port 3 occupies a total of 8 pins, pins 10 through 17. It can be used as input or output. P3 does not need any pull-up resistors, just as P1 and P2 did not. Although port 3 is configured as an input port upon reset, this is not the way it is most commonly used. Port 3 has the additional function of providing some extremely important signals such as interrupts. Table 4-1 provides these alternate functions of P3. This information applies to both 8051 and 8031 chips.

Table 4-1: Port 3 Alternate Functions

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1	TxD	11
P3.2	$\overline{\text{INT0}}$	12
P3.3	$\overline{\text{INT1}}$	13
P3.4	T0	14
P3.5	T1	15
P3.6	$\overline{\text{WR}}$	16
P3.7	$\overline{\text{RD}}$	17

P3.0 and P3.1 are used for the RxD and TxD serial communications signals. See Chapter 10 to see how they are connected. Bits P3.2 and P3.3 are set aside for external interrupts, and are discussed in Chapter 11. Bits P3.4 and P3.5 are used for timers 0 and 1, and are discussed in Chapter 9 where timers are discussed. Finally, P3.6 and P3.7 are used to provide the $\overline{\text{WR}}$ and $\overline{\text{RD}}$ signals of external memories connected in 8031-based systems. Chapter 14 discusses how they are used in 8031-based systems. In systems based on the 8751, 89C51, or DS89C4x0, pins 3.6 and 3.7 are used for I/O while the rest of the pins in port 3 are normally used in the alternate function role.

Example 4-1

Write a test program for the DS89C420/30 chip to toggle all the bits of P0, P1, and P2 every 1/4 of a second. Assume a crystal frequency of 11.0592 MHz.

Solution:

```
;Tested for the DS89C420/30 with XTAL = 11.0592 MHz.

           ORG    0
BACK:      MOV    A,#55H
           MOV    P0,A
           MOV    P1,A
           MOV    P2,A
           ACALL  QSDELAY          ;Quarter of a second delay
           MOV    A,#0AAH
           MOV    P0,A
           MOV    P1,A
           MOV    P2,A
           ACALL  QSDELAY
           SJMP   BACK
;-----------1/4 SECOND DELAY
QSDELAY:
           MOV    R5, #11
H3:        MOV    R4, #248
H2:        MOV    R3, #255
H1:        DJNZ   R3, H1           ;4 MC for DS89C4x0
           DJNZ   R4, H2
           DJNZ   R5, H3
           RET
           END
```

Delay = 11 × 248 × 255 × 4 MC × 90 ns = 250,430 µs

Use an oscilloscope to verify the delay size.

Different ways of accessing the entire 8 bits

In the following code, as in many previous I/O examples, the entire 8 bits of port 1 are accessed.

```
BACK:      MOV    A,#55H
           MOV    P1,A
           ACALL  DELAY
           MOV    A,#0AAH
           MOV    P1,A
           ACALL  DELAY
           SJMP   BACK
```

The above code toggles every bit of P1 continuously. We have seen a variation of the above program before. Now we can rewrite the above code in a more efficient manner by accessing the port directly without going through the accumulator. This is shown next.

```
BACK:      MOV    P1,#55H
           ACALL  DELAY
           MOV    P1,#0AAH
           ACALL  DELAY
           SJMP   BACK
```

The following is another way of doing the same thing.

```
           MOV    A,#55H     ;A=55 HEX
BACK:      MOV    P1,A
           ACALL  DELAY
           CPL    A          ;complement reg. A
           SJMP   BACK
```

We can write another variation of the above code by using a technique called *read-modify-write*. This is shown at the end of this chapter.

Ports status upon reset

Upon reset all ports have value FFH on them as shown in Table 4-2. This makes them input ports upon reset.

Table 4-2: Reset Value of Some 8051 Ports

Register	Reset Value (Binary)
P0	11111111
P1	11111111
P2	11111111
P3	11111111

Review Questions

1. There are a total of _____ ports in the 8051 and each has _____ bits.
2. True or false. All of the 8051 ports can be used for both input and output.
3. Which 8051 ports need pull-up resistors to function as an I/O port?
4. True or false. Upon power-up, the I/O pins are configured as output ports.
5. Show simple statements to send 99H to ports P1 and P2.

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

In this section we further examine 8051 I/O instructions. We pay special attention to I/O bit manipulation since it is a powerful and widely used feature of the 8051 family.

I/O ports and bit-addressability

Sometimes we need to access only 1 or 2 bits of the port instead of the entire 8 bits. A powerful feature of 8051 I/O ports is their capability to access individual bits of the port without altering the rest of the bits in that port. Of the four 8051 ports, we can access either the entire 8 bits or any single bit without altering the rest. When accessing a port in single-bit manner, we use the syntax "`SETB X.Y`" where X is the port number 0, 1, 2, or 3, and Y is the desired bit number from 0 to 7 for data bits D0 to D7. For example, "`SETB P1.5`" sets high bit 5 of port 1. Remember that D0 is the LSB and D7 is the MSB. For example, the following code toggles bit P1.2 continuously.

```
BACK:      CPL   P1.2       ;complement P1.2 only
           ACALL DELAY
           SJMP  BACK

;another variation of the above program follows
AGAIN:     SETB  P1.2       ;change only P1.2=high
           ACALL DELAY
           CLR   P1.2       ;change only P1.2=low
           ACALL DELAY
           SJMP  AGAIN
```

Notice that P1.2 is the third bit of P1, since the first bit is P1.0, the second bit is P1.1, and so on. Table 4-3 shows the bits of the 8051 I/O ports. See Example 4-2 for an example of bit manipulation of I/O bits. Notice in Example 4-2 that unused portions of ports 1 and 2 are undisturbed. This single-bit addressability of I/O ports is one of most powerful features of the 8051 microcontroller and is among the reasons that many designers choose the 8051 over other microcontrollers. We will see the use of the bit-addressability of I/O ports in future chapters.

Table 4-3: Single-Bit Addressability of Ports

P0	P1	P2	P3	Port Bit
P0.0	P1.0	P2.0	P3.0	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7	P1.7	P2.7	P3.7	D7

Example 4-2

Write the following programs.
(a) Create a square wave of 50% duty cycle on bit 0 of port 1.
(b) Create a square wave of 66% duty cycle on bit 3 of port 1.

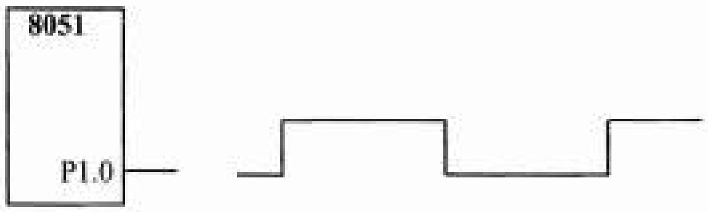
Solution:

(a) The 50% duty cycle means that the "on" and "off" states (or the high and low portions of the pulse) have the same length. Therefore, we toggle P1.0 with a time delay in between each state.

```
HERE:     SETB  P1.0     ;set to high bit 0 of port 1
          LCALL DELAY    ;call the delay subroutine
          CLR   P1.0     ;P1.0=0
          LCALL DELAY
          SJMP  HERE     ;keep doing it
```

Another way to write the above program is:

```
HERE:     CPL   P1.0     ;complement bit 0 of port 1
          LCALL DELAY    ;call the delay subroutine
          SJMP  HERE     ;keep doing it
```



(b) The 66% duty cycle means the "on" state is twice the "off" state.

```
BACK:     SETB  P1.3     ;set port 1 bit 3 high
          LCALL DELAY    ;call the delay subroutine
          LCALL DELAY    ;call the delay subroutine again
          CLR   P1.3     ;clear bit 2 of port 1(P1.3=low)
          LCALL DELAY    ;call the delay subroutine
          SJMP  BACK     ;keep doing it
```

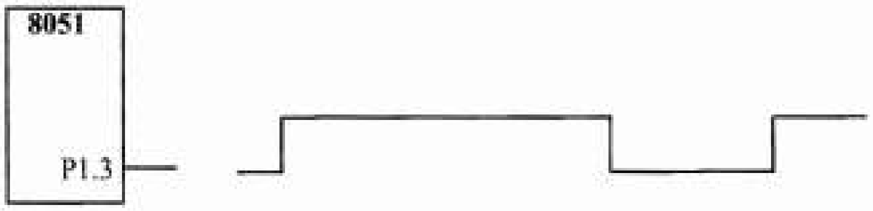


Table 4-4: Single-Bit Instructions

Instruction		Function
SETB	bit	Set the bit (bit = 1)
CLR	bit	Clear the bit (bit = 0)
CPL	bit	Complement the bit (bit = NOT bit)
JB	bit,target	Jump to target if bit = 1 (jump if bit)
JNB	bit,target	Jump to target if bit = 0 (jump if no bit)
JBC	bit,target	Jump to target if bit = 1, clear bit (jump if bit, then clear)

Table 4-4 lists the single-bit instructions for the 8051. We will see the use of these instructions throughout future chapters.

Checking an input bit

The JNB (jump if no bit) and JB (jump if bit = 1) instructions are also widely used single-bit operations. They allow you to monitor a bit and make a decision depending on whether it is 0 or 1. Instructions JNB and JB can be used for any bits of I/O ports 0, 1, 2, and 3, since all ports are bit-addressable. However, most of port 3 is used for interrupts and serial communication signals, and typically is not used for any I/O, either single-bit or byte-wise. This is discussed in Chapters 10 and 11. Table 4-5 shows a list of instructions for reading the ports.

Example 4-3

Write a program to perform the following:
(a) keep monitoring the P1.2 bit until it becomes high
(b) when P1.2 becomes high, write value 45H to port 0
(c) send a high-to-low (H-to-L) pulse to P2.3

Solution:

```
          SETB P1.2           ;make P1.2 an input
          MOV  A,#45H         ;A=45H
AGAIN:    JNB  P1.2,AGAIN     ;get out when P1.2=1
          MOV  P0,A           ;issue A to P0
          SETB P2.3           ;make P2.3 high
          CLR  P2.3           ;make P2.3 low for H-to-L
```

In this program, instruction "`JNB P1.2,AGAIN`" (JNB means jump if no bit) stays in the loop as long as P1.2 is low. When P1.2 becomes high, it gets out of the loop, writes the value 45H to port 0, and creates an H-to-L pulse by the sequence of instructions SETB and CLR.

Table 4-5: Instructions For Reading an Input Port

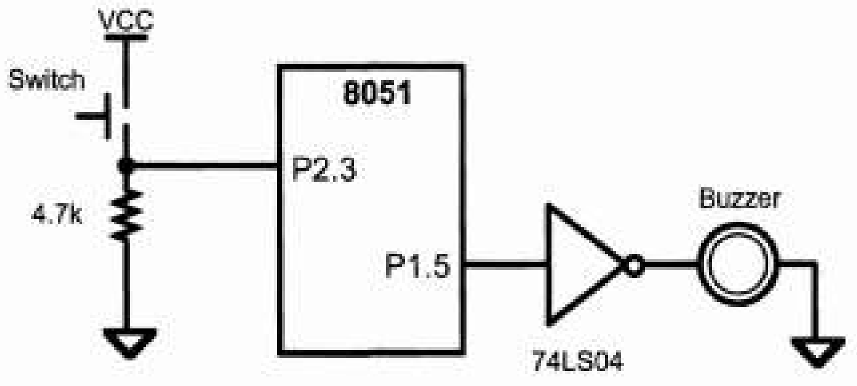
Mnemonic	Example	Description
`MOV A, PX`	`MOV A, P2`	Bring into A the data at P2 pins
`JNB PX.Y,..`	`JNB P2.1, TARGET`	Jump if pin P2.1 is low
`JB PX.Y,..`	`JB P1.3, TARGET`	Jump if pin P1.2 is high
`MOV C, PX.Y`	`MOV C, P2.4`	Copy status of pin P2.4 to CY

Example 4-4

Assume that bit P2.3 is an input and represents the condition of an oven. If it goes high, it means that the oven is hot. Monitor the bit continuously. Whenever it goes high, send a high-to-low pulse to port P1.5 to turn on a buzzer.

Solution:

```
HERE:JNB  P2.3,HERE      ;keep monitoring for high
     SETB P1.5           ;set bit P1.5=1
     CLR  P1.5           ;make high-to-low
     SJMP HERE           ;keep repeating
```



Example 4-5

A switch is connected to pin P1.7. Write a program to check the status of SW and perform the following:
(a) If SW=0, send letter 'N' to P2.
(b) If SW=1, send letter 'Y' to P2.

Solution:

```
          SETB P1.7          ;make P1.7 an input
AGAIN:    JB   P1.2,OVER     ;jump if P1.7=1
          MOV  P2,#'N'       ;SW=0, issue 'N' to P2
          SJMP AGAIN         ;keep monitoring
OVER:     MOV  P2,#'Y'       ;SW=1, issue 'Y' to P2
          SJMP AGAIN         ;keep monitoring
```

Reading a single bit into the carry flag

We can also use the carry flag to save or examine the status of a single bit of the port. To do that, we use the instruction "`MOV C, Px.y`" as shown in the next two examples.

Example 4-6

A switch is connected to pin P1.7. Write a program to check the status of the switch and perform the following:
(a) If switch = 0, send letter 'N' to P2.
(b) If switch = 1, send letter 'Y' to P2.

Use the carry flag to check the switch status. This is a repeat of the last example.

Solution:

```
          SETB P1.7             ;make P1.7 an input
AGAIN:    MOV  C,P1.2           ;read the SW status into CF
          JC   OVER             ;jump if SW = 1
          MOV  P2,#'N'          ;SW = 0, issue 'N' to P2
          SJMP AGAIN            ;keep monitoring
OVER:     MOV  P2,#'Y'          ;SW = 1, issue 'Y' to P2
          SJMP AGAIN            ;keep monitoring
```

Example 4-7

A switch is connected to pin P1.0 and an LED to pin P2.7. Write a program to get the status of the switch and send it to the LED.

Solution:

```
          SETB P1.7             ;make P1.7 an input
AGAIN:    MOV  C,P1.0           ;read the SW status into CF
          MOV  P2.7,C           ;send the SW status to LED
          SJMP AGAIN            ;keep repeating
```

Note: The instruction "`MOV P2.7, P1.0`" is wrong since such an instruction does not exist. However, "`MOV P2, P1`" is a valid instruction.

Notice in Examples 4-6 and 4-7 how the carry flag is used to get a bit of data from the port.

Reading input pins vs. port latch

In reading a port, some instructions read the status of port pins while others read the status of an internal port latch. Therefore, when reading ports there are two possibilities:

1. Read the status of the input pin.
2. Read the internal latch of the output port.

We must make a distinction between these two categories of instructions since confusion between them is a major source of errors in 8051 programming, especially where external hardware is concerned. We discuss these instructions briefly. However, readers must study and understand the material on this topic and on the internal working of ports that is given in Appendix C.2.

Instructions for reading input ports

As stated earlier, to make any bit of any 8051 port an input port, we must write 1 (logic high) to that bit. After we configure the port bits as input, we can use only certain instructions in order to get the external data present at the pins into the CPU. Table 4-6 shows the list of such instructions.

Reading latch for output port

Some instructions read the contents of an internal port latch instead of reading the status of an external pin. Table 4-6 provides a list of these instructions. For example, look at the "`ANL P1, A`" instruction. The sequence of actions taken when such an instruction is executed is as follows.

1. The instruction reads the internal latch of the port and brings that data into the CPU.
2. This data is ANDed with the contents of register A.
3. The result is rewritten back to the port latch.
4. The port pin data is changed and now has the same value as the port latch.

Table 4-6: Instructions Reading a Latch (Read-Modify-Write)

Mnemonic	Example
`ANL  Px`	`ANL  P1,A`
`ORL  Px`	`ORL  P2,A`
`XRL  Px`	`XRL  P0,A`
`JBC  PX.Y,TARGET`	`JBC  P1.1,TARGET`
`CPL  PX.Y`	`CPL  P1.2`
`INC  Px`	`INC  P1`
`DEC  Px`	`DEC  P2`
`DJNZ PX.Y,TARGET`	`DJNZ P1,TARGET`
`MOV  PX.Y,C`	`MOV  P1.2,C`
`CLR  PX.Y`	`CLR  P2.3`
`SETB PX.Y`	`SETB P2.3`

Note: x is 0, 1, 2, or 3 for P0 - P3.

From the above discussion, we conclude that the instructions that read the port latch normally read a value, perform an operation (and possibly change it), then rewrite it back to the port latch. This is often called "*Read-Modify-Write*".

Read-modify-write feature

The ports in the 8051 can be accessed by the read-modify-write technique. This feature saves many lines of code by combining in a single instruction all three actions of (1) reading the port, (2) modifying its value, and (3) writing to the port. The following code first places 01010101 (binary) into port 1. Next, the instruction "`XLR  P1,#0FFH`" performs an XOR logic operation on P1 with 1111 1111 (binary), and then writes the result back into P1.

```
          MOV   P1,#55H        ;P1 = 01010101
AGAIN:    XLR   P1,#0FFH       ;EX-OR P1 with 11111111
          ACALL DELAY
          SJMP  AGAIN
```

Notice that the XOR of 55H and FFH gives AAH. Likewise, the XOR of AAH and FFH gives 55H. Logic instructions are discussed in Chapter 6.

Review Questions

1. True or false. The instruction "`SETB  P2.1`" makes pin P2.1 high while leaving other bits of P2 unchanged.
2. Show one way to toggle the pin P1.7 continuously using 8051 instructions.
3. Using the instruction "`JNB  P2.5,HERE`" assumes that bit P2.5 is an _________ (input, output).
4. Write instructions to get the status of P2.7 and put it on P2.0.
5. Write instructions to toggle both bits of P1.7 and P1.0 continuously.

CAUTION

We strongly recommend that you study Section C.2 (Appendix C) if you are connecting any external hardware to your 8051 system. Failure to use the right instruction or the right connection to port pins can damage the ports of your 8051 system.

SUMMARY

This chapter focused on the I/O ports of the 8051. The four ports of the 8051, P0, P1, P2, and P3, each use 8 pins, making them 8-bit ports. These ports can be used for input or output. Port 0 can be used for either address or data. Port 3 can be used to provide interrupt and serial communication signals. Then I/O instructions of the 8051 were explained, and numerous examples were given. We also showed the bit-addressability of the 8051 ports.

PROBLEMS

SECTION 4.1: 8051 I/O PROGRAMMING

1. The 8051 DIP package is a ____-pin package.
2. Which pins are assigned to V_{CC} and GND?
3. In the 8051, how many pins are designated as I/O port pins?
4. How many pins are designated as P0 and which number are they in the DIP package?
5. How many pins are designated as P1 and which number are they in the DIP package?
6. How many pins are designated as P2 and which number are they in the DIP package?
7. How many pins are designated as P3 and which number are they in the DIP package?
8. Upon RESET, all the bits of ports are configured as _____ (input, output).
9. In the 8051, which port needs a pull-up resistor in order to be used as I/O?
10. Which port of the 8051 does not have any alternate function and can be used solely for I/O?
11. Write a program to get 8-bit data from P1 and send it to ports P0, P2, and P3.
12. Write a program to get 8-bit data from P2 and send it to ports P0 and P1.
13. In P3, which pins are for RxD and TxD?
14. At what memory location does the 8051 wake up upon RESET? What is the implication of that?
15. Write a program to toggle all the bits of P1 and P2 continuously
    (a) using AAH and 55H (b) using the CPL instruction.

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

16. Which ports of the 8051 are bit-addressable?
17. What is the advantage of bit-addressability for 8051 ports?
18. When P1 is accessed as a single-bit port, it is designated as ______.
19. Is the instruction "`CPL P1`" a valid instruction?
20. Write a program to toggle P1.2 and P1.5 continuously without disturbing the rest of the bits.
21. Write a program to toggle P1.3, P1.7, and P2.5 continuously without disturbing the rest of the bits.
22. Write a program to monitor bit P1.3. When it is high, send 55H to P2.
23. Write a program to monitor the P2.7 bit. When it is low, send 55H and AAH to P0 continuously.
24. Write a program to monitor the P2.0 bit. When it is high, send 99H to P1. If it is low, send 66H to P1.
25. Write a program to monitor the P1.5 bit. When it is high, make a low-to-high-to-low pulse on P1.3.
26. Write a program to get the status of P1.3 and put it on P1.4.
27. The P1.4 refers to which bit of P1?
28. Write a program to get the status of P1.7 and P1.6 and put them on P1.0 and P1.7, respectively.

ANSWERS TO REVIEW QUESTIONS

SECTION 4.1: 8051 I/O PROGRAMMING

1. 4, 8.
2. True
3. P0
4. False
5.
```
MOV P1,#99H
MOV P2,#99H
```

SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING

1. True

2.
```
H1: CPL P1.7
    SJMP H1
```

3. Input

4.
```
MOV C,P2.7
MOV P2.0,C
```

5.
```
H1: CPL P1.7
    CPL P1.0
    SJMP H1
```

CHAPTER 7

8051 PROGRAMMING IN C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Examine the C data type for the 8051**
- **>> Code 8051 C programs for time delay and I/O operations**
- **>> Code 8051 C programs for I/O bit manipulation**
- **>> Code 8051 C programs for logic and arithmetic operations**
- **>> Code 8051 C programs for ASCII and BCD data conversion**
- **>> Code 8051 C programs for binary (hex) to decimal conversion**
- **>> Code 8051 C programs to use the 8051 code space**
- **>> Code 8051 C programs for data serialization**

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.

How does the choice of programming language affect the compiled program size? While Assembly language produces a hex file that is much smaller than C, programming in Assembly language is tedious and time consuming. C programming, on the other hand, is less time consuming and much easier to write, but the hex file size produced is much larger than if we used Assembly language. The 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.

The study of C programming for the 8051 is the main topic of this chapter. In Section 7.1, we discuss data types and time delays. I/O programming is shown in Section 7.2. The logic operations AND, OR, XOR, inverter, and shift are discussed in Section 7.3. Section 7.4 describes ASCII and BCD conversions and checksums. In Section 7.5 we show how 8051 C compilers use the program (code) ROM space for data. Finally, in Section 7.6 data serialization for 8051 is shown.

SECTION 7.1: DATA TYPES AND TIME DELAY IN 8051 C

In this section we first discuss C data types for the 8051 and then provide code for time delay functions.

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.

Unsigned char

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, 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 7-1). We can also use the unsigned char data type for a string of ASCII characters, including extended ASCII characters. Example 7-2 shows a string of ASCII characters. See Example 7-3 for toggling ports.

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 7-1

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 7-2

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[]= "012345ABCD";
    unsigned char z;
    for(z=0;z<=10;z++)
      P1=mynum[z];
  }
```

Run the above program on your simulator to see how P1 displays values 30H, 31H, 32H, 33H, 34H, 35H, 41H, 42H, 43H, and 44H, the hex values for ASCII 0, 1, 2, and so on.

Example 7-3

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

Solution:

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

Run the above program on your simulator to see how P1 toggles continuously. Examine the asm code generated by the C compiler.

Signed char

The signed char is an 8-bit data type that uses the most significant bit (D7 of D7 - D0) 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 7-4

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];
  }
```

Run the above program on your simulator to see how P1 displays values of 1, FFH, 2, FEH, 3, FDH, 4, and FCH, the hex values for +1, –1, +2, –2, and so on.

Unsigned int

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. 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. 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. 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. 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. 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*.

Signed int

Signed int is a 16-bit data type that uses the most significant bit (D15 of D15 - D0) 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.

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. As we saw in Chapter 5, some of the SFRs are bit-addressable. Among the SFRs that are widely used and are also bit-addressable are ports P0 - P3. We can use sbit to access the individual bits of the ports as shown in Example 7-5.

Example 7-5

Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 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;
      }
  }
```

Run the above program on your simulator to see how P1.0 toggles continuously.

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 7-1: 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

Time Delay

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. Discussion of the use of the 8051 timer to create time delays is postponed until Chapter 9.

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 IC technology and microprocessor architectural design have seen great advancements. As we saw in Chapter 3, 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. For example, the DS5000 uses 4 clock periods per machine cycle, while the DS89C420 uses only one clock per machine cycle.
2. The crystal frequency connected to the X1 - X2 input pins. The duration of the clock period for the machine cycle is a function of this crystal frequency.
3. 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 subroutine. 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 produces different hex code.

For the above reasons, when we write time delays for C, we must use the oscilloscope to measure the exact duration. Look at Examples 7-6 through 7-8.

Example 7-6

Write an 8051 C program to toggle bits of P1 continuously forever with some delay.

Solution:

```
// Toggle P1 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 7-7

Write an 8051 C program to toggle the bits of P1 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++);
  }
```

Run the above program on your Trainer and use the oscilloscope to measure the delay.

Example 7-8

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++);
  }
```

Review Questions

1. Give the magnitude of the unsigned char and signed char data types.
2. Give the magnitude of the unsigned int and signed int data types.
3. If we are declaring a variable for a person's age, we should use the ___ data type.
4. True or false. Using a for loop to create a time delay is not recommended if you want your code be portable to other 8051 versions.
5. Give three factors that can affect the delay size.

SECTION 7.2: I/O PROGRAMMING IN 8051 C

In this section we look at C programming of the I/O ports for the 8051. We look at both byte and bit I/O programming.

Byte size I/O

As we stated in Chapter 4, ports P0 - P3 are byte-accessible. We use the P0 - P3 labels as defined in the 8051/52 C header file. See Example 7-9. Examine the next few examples to get a better understanding of how ports are accessed in 8051 C.

Example 7-9

LEDs are connected to bits P1 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 7-10

Write an 8051 C program to get a byte of data from P1, 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 7-11

Write an 8051 C program to get a byte of data from P0. If it is less than 100, send it to P1; 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
      }
  }
```

Bit-addressable I/O programming

The I/O ports of P0 - 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 P0 - 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. For example, P1^7 indicates P1.7. When using this method, you need to include the reg51.h file. Study the next few examples to become familiar with the syntax.

Example 7-12

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 7-13

Write an 8051 C program to monitor bit P1.5. If it is high, send 55H to P0; otherwise, send AAH to P2.

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 7-14

A door sensor is connected to the P1.1 pin, and a buzzer is connected to P1.7. 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 7-15

The data pins of an LCD are connected to P1. 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
      }
  }
```

Run the above program on your simulator to see how P1 displays each character of the message. Meanwhile, monitor bit P2.0 after each character is issued.

Accesssing SFR addresses 80 - FFH

Another way to access the SFR RAM space 80 - FFH is to use the sfr data type. This is shown in Example 7-16. We can also access a single bit of any SFR if we specify the bit address as shown in Example 7-17. Both the bit and byte addresses for the P0 - P3 ports are given in Table 7-2. Notice in Examples 7-16 and 7-17, 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 7-2: 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 7-16

Write an 8051 C program to toggle all the bits of P0, P1, and P2 continuously with a 250 ms delay. Use the sfr keyword to declare the port addresses.

Solution:

```
// 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++);
  }
```

Example 7-17

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;
      }
  }
```

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 7-18.

Example 7-18

Write an 8051 C program to get the status of bit P1.0, 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
      }
  }
```

Review Questions

1. The address of P1 is ____________.
2. Write a short program that toggles all bits of P2.
3. Write a short program that toggles only bit P1.0.
4. True or false. The sbit data type is used for both SFR and RAM single-bit addressable locations.
5. True or false. The bit data type is used only for RAM single-bit addressable locations.

SECTION 7.3: LOGIC OPERATIONS IN 8051 C

One of the most important and powerful features of the C language is its ability to perform bit manipulation. Because many books on C do not cover this important topic, it is appropriate to discuss it in this section. This section describes the action of bit-wise logic operators and provides some examples of how they are used.

Bit-wise operators in C

While every C programmer is familiar with the logical operators AND (&&), OR (||), and NOT (!), many C programmers are less familiar with the bit-wise operators AND (&), OR (|), EX-OR (^), Inverter (~), Shift Right (>>), and Shift Left (<<). 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 7-3.

Table 7-3: 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 logical operators.

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

Examples 7-19 and 7-20 show the usage bit-wise operators.

Example 7-19

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

Solution:

```
#include <reg51.h>
void main (void)
  {
    P0= 0x35 & 0x0F;      //ANDing
    P1= 0x04 | 0x68;      //ORing
    P2= 0x54 ^ 0x78;      //XORing
    P0= ~0x55;            //inversing
    P1= 0x9A >> 3;        //shifting right 3 times
    P2= 0x77 >> 4;        //shifting right 4 times
    P0= 0x6 << 4;         //shifting left 4 times
  }
```

Example 7-20

Write an 8051 C program to toggle all the bits of P0 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++);
  }
```

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 */

Study Examples 7-21, 7-22, and 7-23, showing how the bit-wise operators are used in the 8051 C.

Example 7-21

Write an 8051 C program to toggle all the bits of P0, P1, 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 7-22

Write an 8051 C program to get bit P1.0 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 7-23

Write an 8051 C program to read the P1.0 and P1.1 bits and issue an ASCII character to P0 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;
         }
      }
  }
```

Review Questions

1. Find the content of P1 after the following C code in each case.
   (a) `P1=0x37&0xCA;` (b) `P1=0x37|0xCA;` (c) `P1=0x37^0xCA;`
2. To mask certain bits we must AND them with _____.
3. To set high certain bits we must OR them with _____.
4. Ex-ORing a value with itself results in _______.
5. Find the contents of P2 after execution of the following code.

```
P2=0;
P2=P2|0x99;
P2=~P2;
```

SECTION 7.4: DATA CONVERSION PROGRAMS IN 8051 C

Recall that BCD numbers were discussed in Chapter 6. As stated there, many newer microcontrollers have a real-time clock (RTC) where the time and date are kept even when the power is off. Very often the RTC provides the time and date in packed BCD. However, to display them they must be converted to ASCII. In this section we show the 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 7-4.

Table 7-4: 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

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 011 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
```

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 7-24 and 7-25.

Example 7-24

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;
  }
```

Checksum byte in ROM

To ensure the integrity of ROM contents, every system must perform the checksum calculation. 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. To ensure data integrity in ROM, the checksum process uses what is called a *checksum byte*. 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.

1. Add the bytes together and drop the carries.
2. Take the 2's complement of the total sum. This is the checksum byte, which becomes the last byte of the series.

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). To clarify these important concepts, see Example 7-26.

Example 7-26

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 7-27

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
  }
```

Single-step the above program on the 8051 simulator and examine the contents of P1 and P2. Notice that each byte is put on P1 as they are added together.

Example 7-28

Write an 8051 C program to perform step (b) of Example 7-26. If data is good, send ASCII character 'G' to P0. Otherwise send 'B' to P0.

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';
  }
```

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

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.

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. 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. 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, as was shown in Chapter 6. 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 7-29 shows the C program for that algorithm.

Example 7-29

Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on P0, P1, 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;
  }
```

Review Questions

1. For the following decimal numbers, give the packed BCD and unpacked BCD representations.
   (a) 15 (b) 99
2. Show the binary and hex formats for "76" and its BCD version.
3. 67H in BCD when converted to ASCII is ____H and ____H.
4. Does the following convert unpacked BCD in register A to ASCII? `mydata = 0x09 + 0x30;`
5. Why is the use of packed BCD preferable to ASCII?
6. Which one takes more memory space: packed BCD or ASCII?
7. In Question 6, which is more universal?
8. Find the checksum byte for the following values; 22H, 76H, 5FH, 8CH, 99H.
9. To test data integrity, we add them together, including the checksum byte. Then drop the carries. The result must be equal to ____ if the data is not corrupted.
10. An ADC provides an input of 0010 0110. What happens if we output that to the screen?

SECTION 7.5: ACCESSING CODE ROM SPACE IN 8051 C

Using the code (program) space for predefined data is the widely used option in the 8051, as we saw in Chapter 5. In that chapter we saw how to use the Assembly language instruction MOVC to access the data stored in the 8051 code space. In this chapter, we explore the same concept for 8051 C.

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 R0 and R1 registers as we saw in Chapter 5.
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). We can use the "`MOVC A, @A+DPTR`" Assembly language instruction to access it for data (see Chapter 5). There are two problems with using this code space for data. 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. 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. This is discussed next very briefly and we postpone the full discussion to Chapter 14.

3. 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. We will discuss the external memory expansion and how to access it for both Assembly and C in Chapter 14.

Next, we discuss on-chip RAM and ROM space usage by the 8051 C compiler. We have used the Proview32 C compiler to verify the concepts discussed next. Use the compiler of your choice to verify these concepts.

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

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 7-31 the array elements are limited to around 100. Run Example 7-31 on your 8051 C simulator and examine the RAM space allocation. Keep changing the size of the array and monitor the RAM space to see what happens.

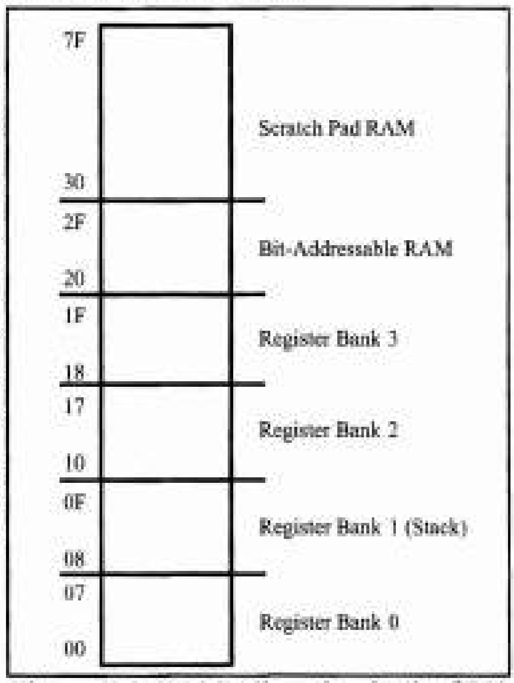


Figure 7-1. RAM Allocation in the 8051

Example 7-30

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];
  }
```

Run the above program on your 8051 simulator and examine the RAM space to locate values 41H, 42H, 43H, 44H, etc., the hex values for ASCII letters 'A', 'B', 'C', and so on.

Example 7-31

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
      }
  }
```

Run the above program on your 8051 simulator and examine the data RAM space to locate values FFH, FEH, FDH, and so on in RAM.

The 8052 RAM data space

Intel added some new features to the 8051 microcontroller and called it the 8052. One of the new features was an extra 128 bytes of RAM space. That means that the 8052 has 256 bytes of RAM space instead of 128 bytes. Remember that the 8052 is code-compatible with the 8051. This means that any program written for the 8051 will run on the 8052, but not the other way around since some features of the 8052 do not exist in the 8051. The extra 128 bytes of RAM helps the 8051/52 C compiler to manage its registers and resources much more effectively. Since the vast majority of the new versions of the 8051 such as DS89C4x0 are really based on 8052 architecture, you should compile your C programs for the 8052 microcontroller. We do that by (1) using the reg52.h header file, and (2) choosing the 8052 option when compiling the program.

Accessing code data space in 8051 C

In all our 8051 C examples so far, byte-size variables were stored in the 128 bytes of RAM. 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 7-32 shows how to use code space for data in 8051 C.

Example 7-32

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];
  }
```

Run the above program on your 8051 simulator and examine the code space to locate values 41H, 42H, 43H, 44H, etc., the hex values for ASCII characters of 'A', 'B', 'C', and so on.

Compiler variations

Look at Example 7-33. It shows three different versions of a program that sends the string "HELLO" to the P1 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. See www.MicroDigitalEd.com for 8051 C compilers.

Example 7-33

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 P1, 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).

See the following Web sites for 8051 C compilers:

www.MicroDigitalEd.com

www.8052.com

Review Questions

1. The 8051 has ____ bytes of data RAM, while the 8052 has ____ bytes.
2. The 8051 has ____ K bytes of code space and ____ K bytes of external data space.
3. True or false. The code space can be used for data but the external data space cannot be used for code.
4. Which space would you use to declare the following values for 8051 C?
   (a) the number of days in the week
   (b) the number of months in a year
   (c) a counter for a delay
5. In 8051 C, we should not use more than 100 bytes of the RAM data space for variables. Why?

SECTION 7.6: DATA SERIALIZATION USING 8051 C

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.

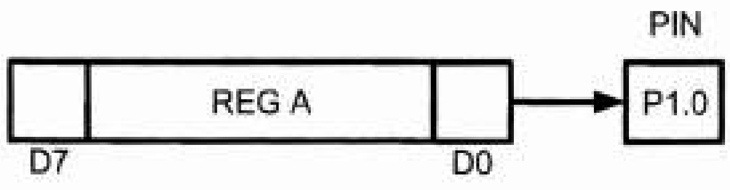
Examine the next four examples to see how data serialization is done in 8051 C.

Example 7-34

Write a C program to send out the value 44H serially one bit at a time via P1.0. 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 7-35

Write a C program to send out the value 44H serially one bit at a time via P1.0. The MSB should go out first.

Solution:

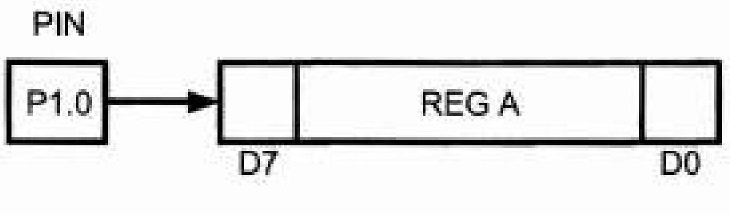
```
//SERIALIZING DATA VIA P1.0 (SHIFTING LEFT)
#include <reg51.h>
sbit P1b0 =  P1^0;
sbit regAMSB = ACC^7;
void main(void)
  {
    unsigned char conbyte = 0x44;
    unsigned char x;
    ACC = conbyte;
    for(x=0; x<8; x++)
      {
        P1b0 = regAMSB;
        ACC = ACC << 1;
      }
  }
```

Example 7-36

Write a C program to bring in a byte of data serially one bit at a time via P1.0. 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;
  }
```



Example 7-37

Write a C program to bring in a byte of data serially one bit at a time via P1.0. The MSB should come in first.

Solution:

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

SUMMARY

This chapter dealt with 8051 C programming, especially I/O programming and time delays in 8051 C. We also showed the logic operators AND, OR, XOR, and complement. In addition, some applications for these operators were discussed. This chapter also described BCD and ASCII formats and conversions in 8051 C. We also compared and contrasted the use of code space and RAM data space in 8051 C. The widely used technique of data serialization was also discussed.

PROBLEMS

SECTION 7.1: DATA TYPES AND TIME DELAY IN 8051 C

1. Indicate what data type you would use for each of the following variables:
   (a) the temperature
   (b) the number of days in a week
   (c) the number of days in a year
   (d) the number of months in a year
   (e) the counter to keep the number of people getting on a bus
   (f) the counter to keep the number of people going to a class
   (g) an address of 64K bytes RAM space
   (h) the voltage
   (i) a string for a message to welcome people to a building
2. Give the hex value that is sent to the port for each of the following C statements:
   (a) `P1=14;` (b) `P1=0x18;` (c) `P1='A';` (d) `P1=7;`
   (e) `P1=32;` (f) `P1=0x45;` (g) `P1=255;` (h) `P1=0x0F;`
3. Give three factors that can affect time delay code size in the 8051 microcontroller.
4. Of the three factors in Problem 3, which one can be set by the system designer?
5. Can the programmer set the number of clock cycles used to execute an instruction? Explain your answer.
6. Explain why various 8051 C compilers produce different hex file sizes.

SECTION 7.2: I/O PROGRAMMING IN 8051 C

7. What is the difference between the sbit and bit data types?
8. Write an 8051 C program to toggle all bits of P1 every 200 ms.
9. Use your 8051 C compiler to see the shortest time delay that you can produce.
10. Write a time delay function for 100 ms.
11. Write an 8051 C program to toggle only bit P1.3 every 200 ms.
12. Write an 8051 C program to count up P1 from 0 - 99 continuously.

SECTION 7.3: LOGIC OPERATIONS IN 8051 C

13. Indicate the data on the ports for each of the following.
    Note: The operations are independent of each other.

(a) `P1=0xF0&0x45;`	(b) `P1=0xF0&0x56;`
(c) `P1=0xF0^0x76;`	(d) `P2=0xF0&0x90;`
(e) `P2=0xF0^0x90;`	(f) `P2=0xF0|0x90;`
(g) `P2=0xF0&0xFF;`	(h) `P2=0xF0|0x99;`
(i) `P2=0xF0^0xEE;`	(j) `P2=0xF0^0xAA;`

14. Find the contents of the port after each of the following operations.

(a) `P1=0x65&0x76;`	(b) `P1=0x70|0x6B;`
(c) `P2=0x95^0xAA;`	(d) `P2=0x5D&0x78;`
(e) `P2=0xC5|0x12;`	(f) `P0=0x6A^0x6E;`
(g) `P1=0x37|0x26;`	

15. Find the port value after each of the following is executed.

(a) `P1=0x65>>2;`	(b) `P2=0x39<<2;`
(c) `P1=0xD4>>3;`	(d) `P1=0xA7<<2;`

16. Show the C code to swap 0x95 to make it 0x59.
17. Write a C program that finds the number of zeros in an 8-bit data item.
18. A stepper motor uses the following sequence of binary numbers to move the motor. How would you generate them in 8051 C?
    `1100,0110,0011,1001`

SECTION 7.4: DATA CONVERSION PROGRAMS IN 8051 C

19. Write a program to convert the following series of packed BCD numbers to ASCII. Assume that the packed BCD is located in data RAM.
    `76H,87H,98H,43H`
20. Write a program to convert the following series of ASCII numbers to packed BCD. Assume that the ASCII data is located in data RAM.
    `"8767"`
21. Write a program to get an 8-bit binary number from P1, convert it to ASCII, and save the result if the input is packed BCD of 00 - 0x99. Assume P1 has 1000 1001 binary as input.

SECTION 7.5: ACCESSING CODE ROM SPACE IN 8051 C

22. Indicate what memory (embedded, data RAM, or code ROM space) you would use for the following variables:
    (a) the temperature
    (b) the number of days in week
    (c) the number of days in a year
    (d) the number of months in a year
    (e) the counter to keep the number of people getting on a bus
    (f) the counter to keep the number of people going to a class
    (g) an address of 64K bytes RAM space
    (h) the voltage
    (i) a string for a message to welcome people to building

23. Discuss why the total size of your 8051 C variables should not exceed 100 bytes.
24. Why do we use the ROM code space for video game characters and shapes?
25. What is the drawback of using RAM data space for 8051 C variables?
26. What is the drawback of using ROM code space for 8051 C data?
27. Write an 8051 C program to send your first and last names to P2. Use ROM code space.

ANSWERS TO REVIEW QUESTIONS

SECTION 7.1: DATA TYPES AND TIME DELAY IN 8051 C

1. 0 to 255 for unsigned char and –128 to +127 for signed char
2. 0 to 65,535 for unsigned int and –32,768 to +32,767 for signed int
3. Unsigned char
4. True
5. (a) Crystal frequency of 8051 system, (b) 8051 machine cycle timing, and (c) compiler use for 8051 C

SECTION 7.2: I/O PROGRAMMING IN 8051 C

1. 90H
2.
```
#include <reg51.h>
void main()
  {
   P2 = 0x55;
   P2 = 0xAA
  }
```
3.
```
#include <reg51.h>
sbit P10bit = P1^0;
void main()
  {
   P10bit = 0;
   P10bit = 1;
  }
```
4. False, only to SFR bit
5. True

SECTION 7.3: LOGIC OPERATIONS IN 8051 C

1. (a) 02 (b) FFH (c) FDH
2. Zeros
3. One
4. All zeros
5. 66H

SECTION 7.4: DATA CONVERSION PROGRAMS IN 8051 C

1. (a) 15H = 0001 0101 packed BCD, 0000 0001,0000 0101 unpacked BCD
   (b) 99H = 1001 1001 packed BCD, 0000 1001,0000 1001 unpacked BCD
2. 3736H = 00110111 00110110B
   and in BCD we have 76H = 0111 0110B

3. 36H, 37H
4. Yes, since A = 39H
5. Space savings
6. ASCII 7. ASCII
8. 21CH
9. 00
10. First convert from binary to decimal, then to ASCII, then send to screen.

SECTION 7.5: ACCESSING CODE ROM SPACE IN 8051 C

1. 128, 256
2. 64K, 64K
3. True
4. (a) data space, (b) data space, (c) RAM space
5. The compiler starts storing variables in code space.

CHAPTER 9

8051 TIMER PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

**>>	List the timers of the 8051 and their associated registers**
**>>	Describe the various modes of the 8051 timers**
**>>	Program the 8051 timers in Assembly and C to generate time delays**
**>>	Program the 8051 counters in Assembly and C as event counters**

The 8051 has two timers/counters. They can be used either as timers to generate a time delay or as counters to count events happening outside the microcontroller. In Section 9.1 we see how these timers are used to generate time delays. In Section 9.2 we show how they are used as event counters. In Section 9.3 we use C language to program the 8051 timers.

SECTION 9.1: PROGRAMMING 8051 TIMERS

The 8051 has two timers: Timer 0 and Timer 1. They can be used either as timers or as event counters. In this section we first discuss the timers' registers and then show how to program the timers to generate time delays.

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, R0, R1, R2, etc. For example, the instruction "`MOV TL0,#4FH`" moves the value 4FH into TL0, the low byte of Timer 0. These registers can also be read like any other register. For example, "`MOV R5,TH0`" saves TH0 (high byte of Timer 0) in R5.

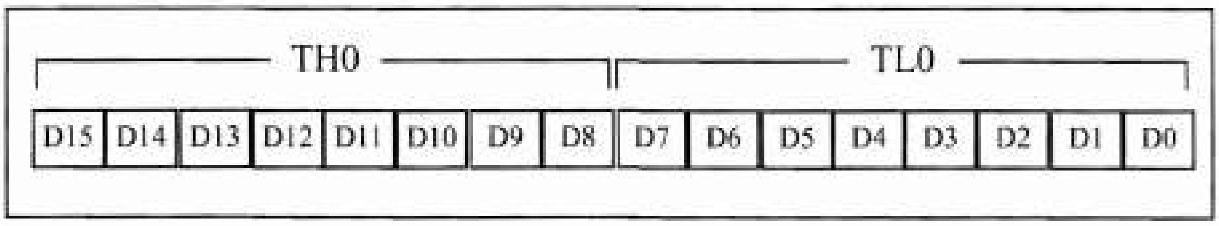


Figure 9-1. Timer 0 Registers

Timer 1 registers

Timer 1 is also 16 bits, and its 16-bit register is split into two bytes, referred to as TL1 (Timer 1 low byte) and TH1 (Timer 1 high byte). These registers are accessible in the same way as the registers of Timer 0.

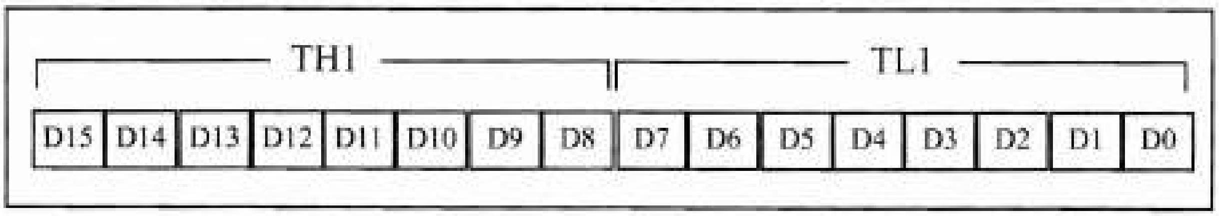


Figure 9-2. Timer 1 Registers

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 set aside 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. These options are discussed next.

(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

Figure 9-3. TMOD Register

M1, M0

M0 and M1 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. This section is concerned with this choice. The timer's use as an event counter is discussed in the next section.

Example 9-1

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:

```
(a) TMOD = 00000001, mode 1 of Timer 0 is selected.
(b) TMOD = 00100000, mode 2 of Timer 1 is selected.
(c) TMOD = 00010010, mode 2 of Timer 0, and mode 1 of
    Timer 1 are selected.
```

Clock source for timer

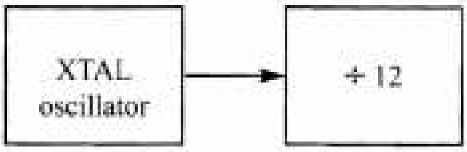
As you know, every timer needs a clock pulse to tick. What is the source of the clock pulse for the 8051 timers? If C/T = 0, the crystal frequency attached to the 8051 is the source of the clock for the timer. This means that the size of the crystal frequency attached to the 8051 also decides the speed at which the 8051 timer ticks. The frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051. See Example 9-2.

Example 9-2

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
```

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

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, as we will see in Chapter 10.

GATE

The other bit of the TMOD register is the GATE bit. Notice in the TMOD register of Figure 9-3 that both Timers 0 and 1 have the GATE bit. What is its purpose? 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. 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. 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`". The use of external hardware to stop or start the timer is discussed in Chapter 11 when interrupts are discussed.

Example 9-3

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.
```

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.

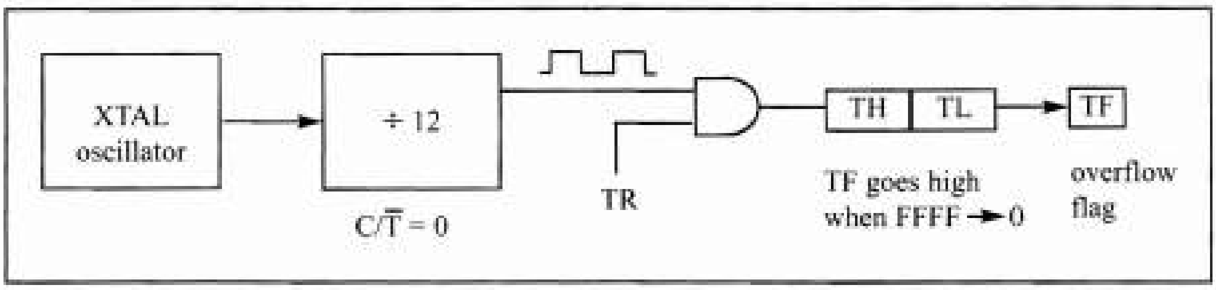
Mode 1 programming

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 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 TR0`" or "`CLR TR1`", for Timer 0 and Timer 1, respectively. Again, it must be noted that each timer has its own timer flag: TF0 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 9-4.

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.

To calculate the exact time delay and the square wave frequency generated on pin P1.5, we need to know the XTAL frequency. See Example 9-5.

From Example 9-6 we can develop a formula for delay calculations using mode 1 (16-bit) of the timer for a crystal frequency of XTAL = 11.0592 MHz. This is given in Figure 9-4. 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	**(b) in decimal**
(FFFF - YYXX + 1) × 1.085 µs where YYXX are TH, TL initial values respectively. Notice that values YYXX are in hex.	Convert YYXX values of the TH,TL register to decimal to get a NNNNN decimal number, then(65536 - NNNNN) × 1.085 µs

Figure 9-4. Timer Delay Calculation for XTAL = 11.0592 MHz

Example 9-4

In the following program, we are creating a square wave of 50% duty cycle (with equal portions high and low) on the P1.5 bit. Timer 0 is used to generate the time delay. Analyze the program.

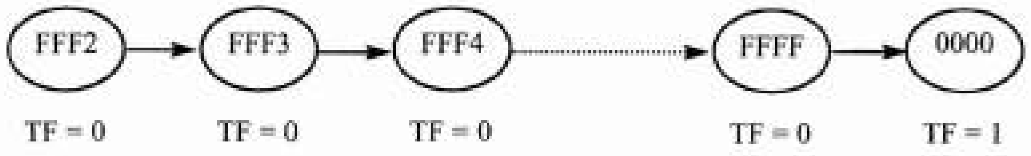
```
              MOV    TMOD,#01          ;Timer 0, mode 1(16-bit mode)
HERE:         MOV    TL0,#0F2H         ;TL0 = F2H, the Low byte
              MOV    TH0,#0FFH         ;TH0 = FFH, the High byte
              CPL    P1.5              ;toggle P1.5
              ACALL DELAY
              SJMP   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 TH0 - TL0.
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 TR0`" 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 (TF0 = 1). At that point, the JNB instruction falls through.
7. Timer 0 is stopped by the instruction "`CLR TR0`". 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 9-5

In Example 9-4, 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 µs. In other words, Timer 0 counts up each 1.085 µs resulting in delay = number of counts × 1.085 µs.
The number of counts for the rollover is FFFFH – FFF2H = 0DH (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 × 1.085 µs = 15.19 µs for half the pulse. For the entire period T = 2 × 15.19 µs = 30.38 µs gives us the time delay generated by the timer.

Example 9-6

In Example 9-5, 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-1 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 × 28 × 1.085 µs = 60.76 µs and F = 16458.2 Hz.

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

Example 9-7

Find the delay generated by Timer 0 in the following code, using both of the methods of Figure 9-4. 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:

(a) (FFFF - B83E + 1) = 47C2H = 18370 in decimal and 18370 × 1.085 μs = 19.93145 ms.

(b) 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 μs in duration. Therefore, we have 18370 × 1.085 μs = 19.93145 ms as the width of the pulse.

Example 9-8

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) × 1.085 μs = 71.1065 ms.

Example 9-9

The following program generates a square wave on pin P1.5 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 × 1.085 µs = 38.274 ms for half of the square wave. The entire square wave length is 38.274 × 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.

In Examples 9-7 and 9-8, we did not reload TH and TL since it was a single pulse. Look at Example 9-9 to see how the reloading works in mode 1.

Finding values to be loaded into the timer

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.

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 µs.
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 9-10

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 µs. This means that out of many 1.085 µs intervals we must make a 5 ms pulse. To get that, we divide one by the other. We need 5 ms / 1.085 µs = 4608 clocks. To achieve that we need to load into TL and TH the value 65536 – 4608 = 60928 = EE00H. Therefore, we have TH = EE and TL = 00.

```
            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 9-11

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

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.

(a) T = 1 / f = 1 / 2 kHz = 500 µs the period of the square wave.

(b) 1/2 of it for the high and low portions of the pulse is 250 µs.

(c) 250 µs / 1.085 µs = 230 and 65536 – 230 = 65306, which in hex is FF1AH.

(d) 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 9-12

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.
(a) T = 1 / 50 Hz = 20 ms, the period of the square wave.
(b) 1/2 of it for the high and low portions of the pulse = 10 ms
(c) 10 ms / 1.085 μs = 9216 and 65536 - 9216 = 56320 in decimal, and in hex it is DC00H.
(d) 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
```

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. What if that is not enough? Example 9-13 shows how to achieve large time delays.

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 μs. 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).
5. 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 9-13

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 × 1.085 μs = 70.820 ms, and for 200 of them we have 200 × 70.820 ms = 14.164024 seconds.

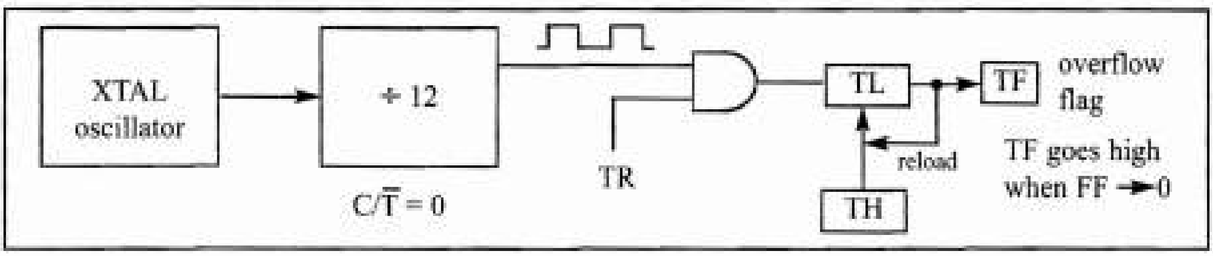
Mode 0

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.

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 TR0`" for Timer 0 and "`SETB TR1`" 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, TF0 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.

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, as we will see in Chapter 10.

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 9-14 illustrates these points. To achieve a larger delay, we can use multiple registers as shown in Example 9-15.

Example 9-14

Assuming that XTAL = 11.0592 MHz, find (a) the frequency of the square wave generated on pin P1.0 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:

(a) First notice the target address of SJMP. In mode 2 we do not need to reload TH since it is auto-reload. Now $(256 - 05) \times 1.085\ \mu s = 251 \times 1.085\ \mu s = 272.33\ \mu s$ 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 $T = 2 \times 272.33\ \mu s = 544.67\ \mu s$ and the frequency = 1.83597 kHz.

(b) To get the smallest frequency, we need the largest T and that is achieved when TH = 00. In that case, we have $T = 2 \times 256 \times 1.085\ \mu s = 555.52\ \mu s$ and the frequency = 1.8 kHz.

Example 9-15

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
```

$T = 2\,(250 \times 256 \times 1.085\ \mu s) = 138.88$ ms, and frequency = 72 Hz.

Example 9-16

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

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,#-100`", the assembler will calculate the -100 = 9C, and makes THl = 9C in hex. This makes our job easier.

Example 9-17

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 × 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 µs + 162.25 µs = 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.

In this section, we used the 8051 timer for time delay generation. However, a more powerful and creative use of these timers is to use them as event counters. We discuss this use of the counter next.

Review Questions

1. How many timers do we have in the 8051?
2. Each timer has _____ registers that are ___ bits wide.
3. TMOD register is a(n) ___-bit register.
4. True or false. The TMOD register is a bit-addressable register.
5. Indicate the selection made in the instruction "`MOV TMOD,#20H`".
6. In mode 1, the counter rolls over when it goes from ____ to ____.
7. In mode 2, the counter rolls over when it goes from ____ to ____.
8. In the instruction "`MOV TH1,#-200`", find the hex value for the TH register.
9. To get a 2-ms delay, what number should be loaded into TH, TL using mode 1? Assume that XTAL = 11.0592 MHz.
10. To get a 100-µs delay, what number should be loaded into the TH register using mode 2? Assume XTAL = 11.0592 MHz.

SECTION 9.2: COUNTER PROGRAMMING

In the last section we used the timer/counter of the 8051 to generate time delays. These timers can also be used as counters counting events happening outside the 8051. The use of the timer/counter as an event counter is covered in this section. As far as the use of a timer as an event counter is concerned, everything that we have talked about in programming the timer in the last section also applies to programming it as a counter, except the source of the frequency. When the timer/counter is used as a timer, the 8051's crystal is used as the source of the frequency. When it is used as a counter, however, it is a pulse outside the 8051 that increments the TH, TL registers. In counter mode, notice that the TMOD and TH, TL registers are the same as for the timer discussed in the last section; they even have the same names. The timer's modes are the same as well.

C/T bit in TMOD register

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. 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. These pins are called T0 (Timer 0 input) and T1 (Timer 1 input). Notice that these two pins belong to port 3. 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 9-1: 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 9-18

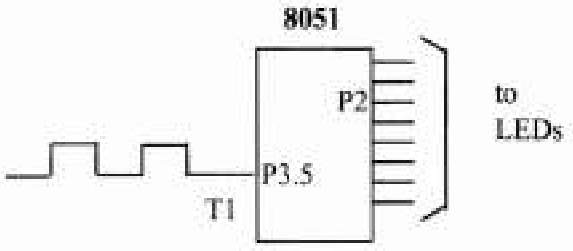
Assuming that clock pulses are fed into pin T1, write a program for counter 1 in mode 2 to count the pulses and display the state of the TL1 count on P2.

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 9-18, 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 9-18, the TL data was displayed in binary. In Example 9-19, the TL registers are converted to ASCII to be displayed on an LCD.

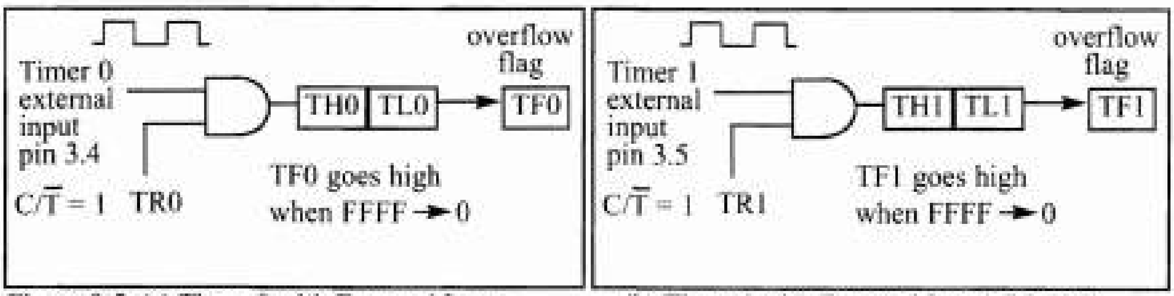

Timer 0 external input pin 3.4
C/T = 1
TR0
TH0
TL0
overflow flag
TF0
TF0 goes high when FFFF → 0
Timer 1 external input pin 3.5
C/T = 1
TR1
TH1
TL1
overflow flag
TF1
TF1 goes high when FFFF → 0

Figure 9-5. (a) Timer 0 with External Input (Mode 1) **(b) Timer 1 with External Input (Mode 1)**

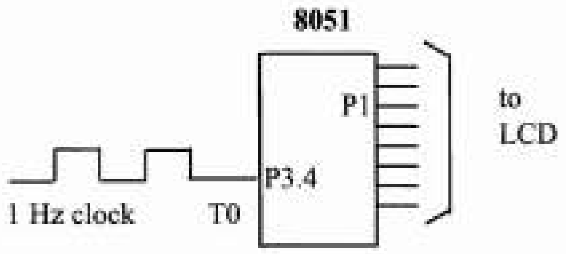
Example 9-19

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 TH0 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, TL0 = 0.
To solve this problem, load TL0 with -60 at the beginning of the program.

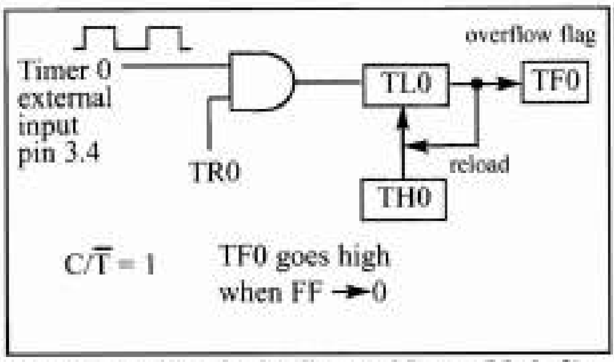


Figure 9-6. Timer 0 with External Input (Mode 2)

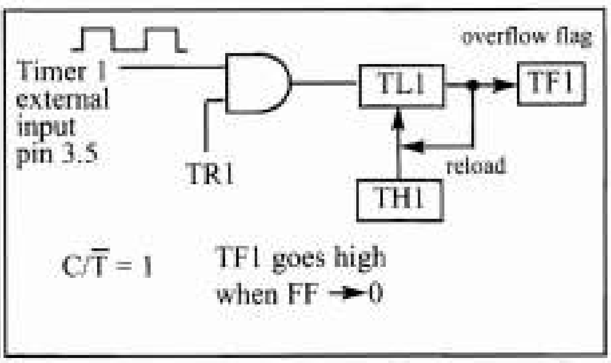


Figure 9-7. Timer 1 with External Input (Mode 2)

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.

Before we finish this chapter, we need to state two important points.

1. You might think that the use of the instruction "`JNB TFx,target`" to monitor the raising of the TFx flag is a waste of the microcontroller's time. You are right. There is a solution to this: the use of interrupts. By using interrupts we can go about doing other things with the microcontroller. When the TF flag is raised it will inform us. This important and powerful feature of the 8051 is discussed in Chapter 11.
2. You might wonder to what register TR0 and TR1 belong. They belong to a register called TCON, which is discussed next.

Table 9-2: 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

TCON register

In the examples so far we have seen the use of the TR0 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. As shown in Table 9-2, 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, which will be discussed in Chapter 11. 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. Table 9-2 shows replacements of some of the instructions we have seen so far.

The case of GATE = 1 in TMOD

Before we finish this section we need to discuss another case of the GATE bit in the TMOD register. All discussion so far has assumed that GATE = 0. When GATE = 0, the timer is started with instructions "`SETB TR0`" and "`SETB TR1`", for Timers 0 and 1, respectively. What happens if the GATE bit in TMOD is set to 1? As can be seen in Figures 9-8 and 9-9, 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. 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. 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 TR0`" 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.

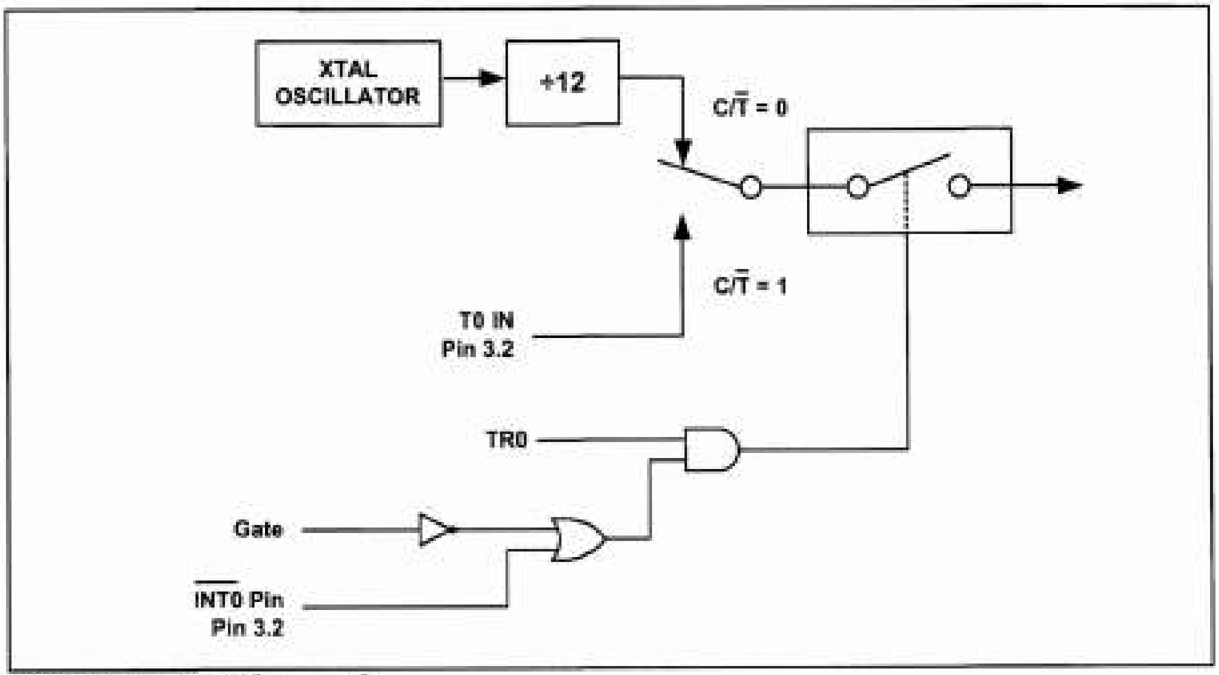


Figure 9-8. Timer/Counter 0

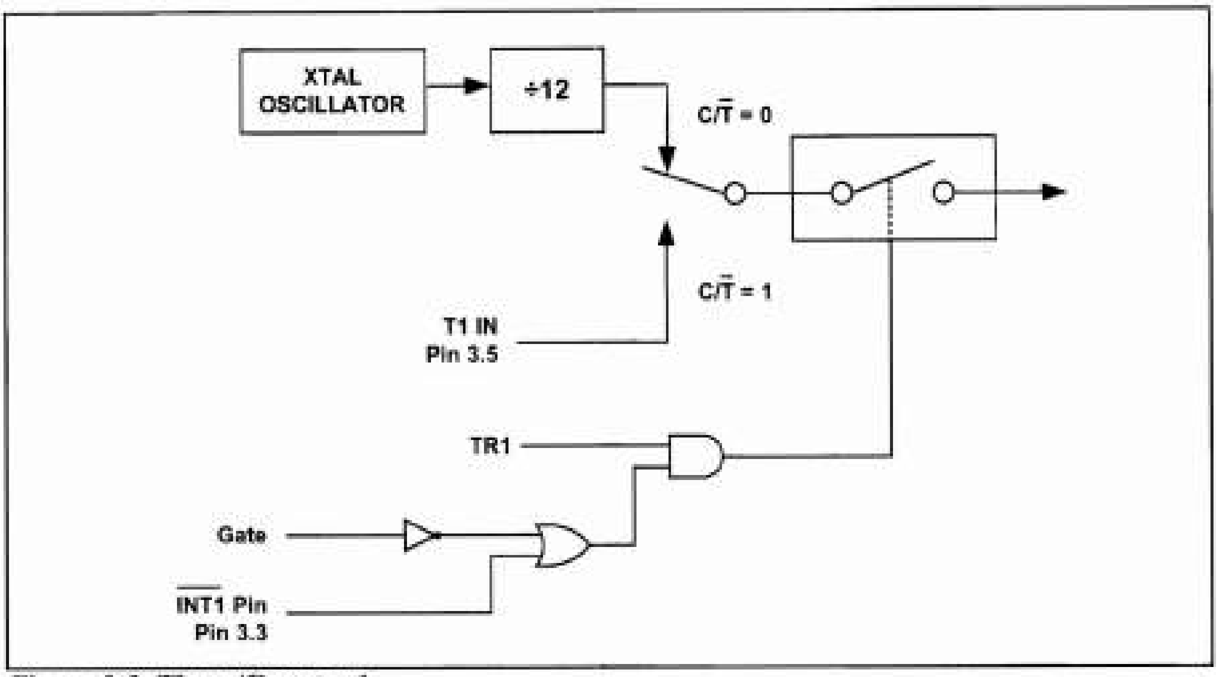


Figure 9-9. Timer/Counter 1

Review Questions

1. Who provides the clock pulses to 8051 timers if C/T = 0?
2. Who provides the clock pulses to 8051 timers if C/T = 1?
3. Does the discussion in Section 9.1 apply to timers if C/T = 1?
4. What must be done to allow P3.4 to be used as an input for T1, and why?
5. What is the equivalent of the following instruction? "`SETB TCON.6`"

SECTION 9.3: PROGRAMMING TIMERS 0 AND 1 IN 8051 C

In Chapter 7 we showed some examples of C programming for the 8051. In this section we study C programming for the 8051 timers. As we saw in the examples in Chapter 7, the general-purpose registers of the 8051, such as R0 - R7, A, and B, are under the control of the C compiler and are not accessed directly by C statements. In the case of SFRs, the entire RAM space of 80 - FFH is accessible directly using 8051 C statements. As an example of accessing the SFRs directly, we saw how to access ports P0 - P3 in Chapter 7. Next, we discuss how to access the 8051 timers directly using C statements.

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. This is shown in Example 9-20. Example 9-20 also shows how to access the TR and TF bits.

Example 9-20

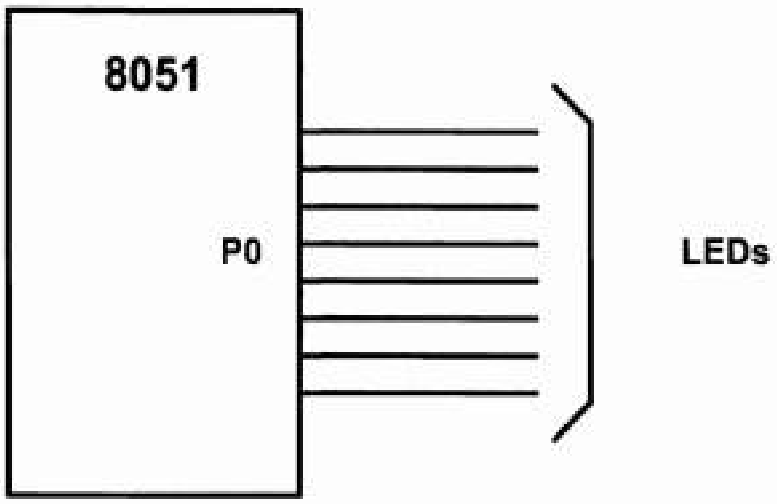
Write a 8051 C program to toggle all the bits of port P1 continuously with some delay in between. Use Timer 0, 16-bit mode to generate the delay.

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

As we mentioned in Chapter 7, the delay length depends on three factors: (a) the crystal frequency, (b) the number of clocks per machine cycle, and (c) the C compiler. The original 8051 used 1/12 of the crystal oscillator frequency as one machine cycle. In other words, each machine cycle is equal to 12 clocks periods of the crystal frequency connected to the X1 - X2 pins. The time it takes for the 8051 to execute an instruction is one or more machine cycles, as shown in Appendix A. To speed up the 8051, many recent versions of the 8051 have reduced the number of clocks per machine cycle from 12 to four, or even one. For example, the AT89C51/52 uses 12, while the DS5000 uses 4 clocks, and the DS89C4x0 uses only one clock per machine cycle. As we mentioned earlier in this chapter, the 8051 timers also use the crystal frequency as the clock source. The frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051, regardless of the 8051 version. In other words, for the AT89C51/52, DS5000, or DS89C4x0 the duration of the time to execute an instruction varies, but they all use 1/12th of the crystal's oscillator frequency for the clock source to the timers. This is done in order to maintain compatibility with the original 8051 since many designers use timers to create time delay. This is an important point and needs to be emphasized. The C compiler is a factor in the delay size since various 8051 C compilers generate different hex code sizes. This explains why the timer delay duration is unknown for Example 9-20 since none of the other factors mentioned is specified.

Delay duration for the AT89C51/52 and DS89C4x0 chips

As we stated before, there is a major difference between the AT89C51 and DS89C4x0 chips in term of the time it takes to execute a single instruction. Although the DS89C4x0 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)

Examples 9-23 through 9-25 shows 8051 C programming of timers 0 and 1 in mode 2 (8-bit auto-reload). Study these examples to get familiar with the syntax.

Example 9-21

Write an 8051 C program to toggle only bit P1.5 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 9-22

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

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

Example 9-23

Write an 8051 C program to toggle only pin P1.5 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
  }
```

$256 - 23 = 233$

$23 \times 1.085\ \mu s = 25\ \mu s$

$25\ \mu s \times 250 \times 40 = 250$ 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 9-24

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:

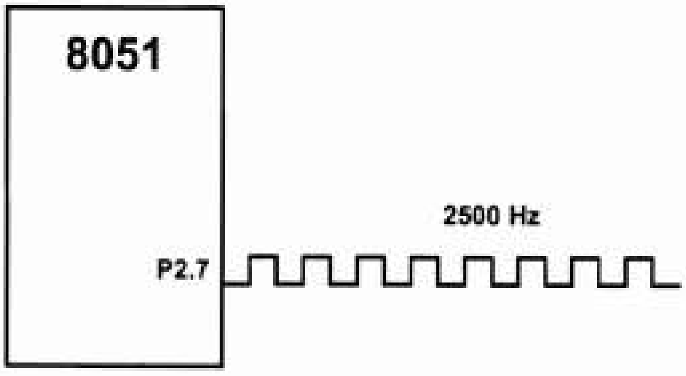
//tested for DS89C420, XTAL = 11.0592 MHz, using the Proview32 compiler

```
#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 9-25

A switch is connected to pin P1.2. Write an 8051 C program to monitor SW and create the following frequencies on pin P1.7:

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

C Programming of timers 0 and 1 as counters

In Section 9.2 we showed how to use timers 0 and 1 as event 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 T0 (P3.4) and T1 (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.

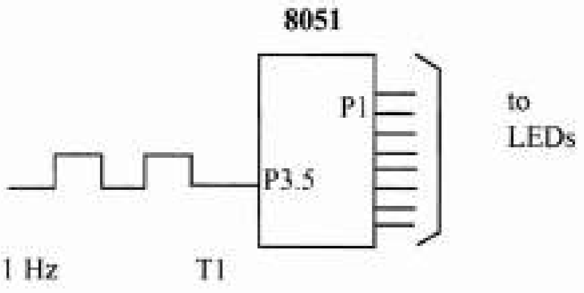
Example 9-26

Assume that a 1-Hz external clock is being fed into pin T1 (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 P1. Start the count at 0H.

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.

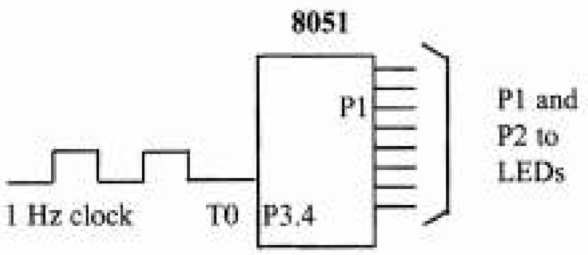


Example 9-27

Assume that a 1-Hz external clock is being fed into pin T0 (P3.4). Write a C program for counter 0 in mode 1 (16-bit) to count the pulses and display the TH0 and TL0 registers on P2 and P1, 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 9-28

Assume that a 2-Hz external clock is being fed into pin T1 (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 P0, P1, and P2 where P0 has the least significant digit. Set the initial value of TH0 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.

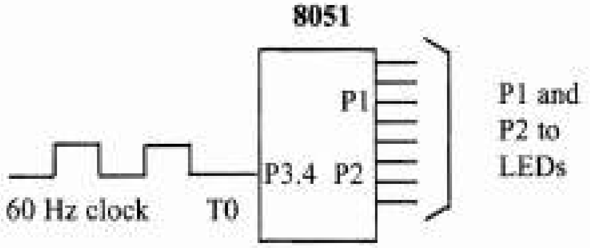
```
#include <reg51.h>
void BinToASCII(unsigned char);
void main()
  {
    unsigned char value;
    T1=1;
    TMOD=0x06;
    TH0=0;

    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 9-29

Assume that a 60-Hz external clock is being fed into pin T0 (P3.4). Write a C program for counter 0 in mode 2 (8-bit auto-reload) to display the seconds and minutes on P1 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.

For Examples of Timer 2, see the www.MicroDigitalEd.com Web site.

Review Questions

1. Who provides the clock pulses to 8051 timers if C/T = 0?
2. Indicate the selection made in the statement "`TMOD = 0x20`".
3. In mode 1, the counter rolls over when it goes from ____ to ____.
4. In mode 2, the counter rolls over when it goes from ____ to ____.
5. In the statement "`TH1 = -200`", find the hex value for the TH register.
6. TF0 and TF1 are part of register ____.
7. In Question 6, is the register bit-addressable?
8. Show how to monitor the TF1 flag for high in 8051 C.

SUMMARY

The 8051 has two timers/counters. When used as timers they can generate time delays. When used as counters they can serve as event counters. This chapter showed how to program the timers/counters for various modes.

The two timers are accessed as two 8-bit registers: TL0 and TH0 for Timer 0, and TL1 and TH1 for Timer 1. Both timers use the TMOD register to set timer operation modes. The lower 4 bits of TMOD are used for Timer 0 and the upper 4 bits are used for Timer 1.

There are different modes that can be used for each timer. Mode 0 sets the timer as a 13-bit timer, mode 1 sets it as a 16-bit timer, and mode 2 sets it as an 8-bit timer.

When the timer/counter is used as a timer, the 8051's crystal is used as the source of the frequency; when it is used as a counter, however, it is a pulse outside the 8051 that increments the TH, TL registers.

PROBLEMS

SECTION 9.1: PROGRAMMING 8051 TIMERS

1. How many timers do we have in the 8051?
2. The timers of the 8051 are ____-bit and are designated as ______ and _______.
3. The registers of Timer 0 are accessed as ______ and _______.

4. The registers of Timer 1 are accessed as ______ and _______.
5. In Questions 3 and 4, are the registers bit-addressable?
6. The TMOD register is a(n) ___-bit register.
7. What is the job of the TMOD register?
8. True or false. TMOD is a bit-addressable register.
9. Find the TMOD value for both Timer 0 and Timer 1, mode 2, software start / stop (gate = 0), with the clock coming from the 8051's crystal.
10. Find the frequency and period used by the timer if the crystal attached to the 8051 has the following values.
    (a) XTAL = 11.0592 MHz (b) XTAL = 20 MHz
    (c) XTAL = 24 MHz (d) XTAL = 30 MHz
11. Indicate the size of the timer for each of the following modes.
    (a) mode 0 (b) mode 1 (c) mode 2
12. Indicate the rollover value (in hex and decimal) of the timer for each of the following modes.
    (a) mode 0 (b) mode 1 (c) mode 2
13. Indicate when the TF1 flag is raised for each of the following modes.
    (a) mode 0 (b) mode 1 (c) mode 2
14. True or false. Both Timer 0 and Timer 1 have their own TF.
15. True or false. Both Timer 0 and Timer 1 have their own timer start (TR).
16. Assuming XTAL = 11.0592 MHz, indicate when the TF0 flag is raised for the following program.

```
MOV   TMOD,#01
MOV   TL0,#12H
MOV   TH0,#1CH
SETB  TR0
```

17. Assuming that XTAL = 16 MHz, indicate when the TF0 flag is raised for the following program.

```
MOV   TMOD,#01
MOV   TL0,#12H
MOV   TH0,#1CH
SETB  TR0
```

18. Assuming that XTAL = 11.0592 MHz, indicate when the TF0 flag is raised for the following program.

```
MOV   TMOD,#01
MOV   TL0,#10H
MOV   TH0,#0F2H
SETB  TR0
```

19. Assuming that XTAL = 20 MHz, indicate when the TF0 flag is raised for the following program.

```
MOV   TMOD,#01
MOV   TL0,#12H
MOV   TH0,#1CH
SETB  TR0
```

20. Assume that XTAL = 11.0592 MHz. Find the TH1,TL1 value to generate a time delay of 2 ms. Timer 1 is programmed in mode 1.

21. Assume that XTAL = 16 MHz. Find the TH1,TL1 value to generate a time delay of 5 ms. Timer 1 is programmed in mode 1.
22. Assuming that XTAL = 11.0592 MHz, program Timer 0 to generate a time delay of 2.5 ms.
23. Assuming that XTAL = 11.0592 MHz, program Timer 1 to generate a time delay of 0.2 ms.
24. Assuming that XTAL = 20 MHz, program Timer 1 to generate a time delay of 100 ms.
25. Assuming that XTAL = 11.0592 MHz, and we are generating a square wave on pin P1.2, find the lowest square wave frequency that we can generate using mode 1.
26. Assuming that XTAL = 11.0592 MHz, and we are generating a square wave on pin P1.2, find the highest square wave frequency that we can generate using mode 1.
27. Assuming that XTAL = 16 MHz, and we are generating a square wave on pin P1.2, find the lowest square wave frequency that we can generate using mode 1.
28. Assuming that XTAL = 16 MHz, and we are generating a square wave on pin P1.2, find the highest square wave frequency that we can generate using mode 1.
29. In mode 2 assuming that TH1 = F1H, indicate which states timer 2 goes through until TF1 is raised. How many states is that?
30. Program Timer 1 to generate a square wave of 1 kHz. Assume that XTAL = 11.0592 MHz.
31. Program Timer 0 to generate a square wave of 3 kHz. Assume that XTAL = 11.0592 MHz.
32. Program Timer 0 to generate a square wave of 0.5 kHz. Assume that XTAL = 20 MHz.
33. Program Timer 1 to generate a square wave of 10 kHz. Assume that XTAL = 20 MHz.
34. Assuming that XTAL = 11.0592 MHz, show a program to generate a 1-second time delay. Use any timer you want.
35. Assuming that XTAL = 16 MHz, show a program to generate a 0.25-second time delay. Use any timer you want.
36. Assuming that XTAL = 11.0592 MHz and that we are generating a square wave on pin P1.3, find the lowest square wave frequency that we can generate using mode 2.
37. Assuming that XTAL = 11.0592 MHz and that we are generating a square wave on pin P1.3, find the highest square wave frequency that we can generate using mode 2.
38. Assuming that XTAL = 16 MHz and that we are generating a square wave on pin P1.3, find the lowest square wave frequency that we can generate using mode 2.
39. Assuming that XTAL = 16 MHz and that we are generating a square wave on pin P1.3, find the highest square wave frequency that we can generate using mode 2.
40. Find the value (in hex) loaded into TH in each of the following.

(a)	MOV TH0,#-12	(b)	MOV TH0,#-22
(c)	MOV TH0,#-34	(d)	MOV TH0,#-92
(e)	MOV TH1,#-120	(f)	MOV TH1,#-104
(g)	MOV TH1,#-222	(h)	MOV TH1,#-67

41. In Problem 40, indicate by what number the machine cycle frequency of 921.6 kHz (XTAL = 11.0592 MHz) is divided.
42. In Problem 41, find the time delay for each case from the time the timer starts to the time the TF flag is raised.

SECTION 9.2: COUNTER PROGRAMMING

43. To use the timer as an event counter we must set the C/T bit in the TMOD register to ________ (low, high).
44. Can we use both of the timers as event counters?
45. For counter 0, which pin is used to input clocks?
46. For counter 1, which pin is used to input clocks?
47. Program Timer 1 to be an event counter. Use mode 1 and display the binary count on P1 and P2 continuously. Set the initial count to 20,000.
48. Program Timer 0 to be an event counter. Use mode 2 and display the binary count on P2 continuously. Set the initial count to 20.
49. Program Timer 1 to be an event counter. Use mode 2 and display the decimal count on P2, P1, and P0 continuously. Set the initial count to 99.
50. The TCON register is a(n) _____-bit register.
51. True or false. The TCON register is not a bit-addressable register.
52. Give another instruction to perform the action of "SETB TR0".

SECTION 9.3: PROGRAMMING TIMERS 0 AND 1 IN 8051 C

53. Program Timer 0 in C to generate a square wave of 3 kHz. Assume that XTAL = 11.0592 MHz.
54. Program Timer 1 in C to generate a square wave of 3 kHz. Assume that XTAL = 11.0592 MHz.
55. Program Timer 0 in C to generate a square wave of 0.5 kHz. Assume that XTAL = 11.0592 MHz.
56. Program Timer 1 in C to generate a square wave of 0.5 kHz. Assume that XTAL = 11.0592 MHz.
57. Program Timer 1 in C to be an event counter. Use mode 1 and display the binary count on P1 and P2 continuously. Set the initial count to 20,000.
58. Program Timer 0 in C to be an event counter. Use mode 2 and display the binary count on P2 continuously. Set the initial count to 20.

ANSWERS TO REVIEW QUESTIONS

SECTION 9.1: PROGRAMMING 8051 TIMERS

1. Two
2. 2, 8
3. 8
4. False
5. 0010 0000 indicates Timer 1, mode 2, software start and stop, and using XTAL for frequency.
6. FFFFH to 0000
7. FFH to 00
8. –200 is 38H; therefore, TH1 = 38H
9. 2 ms/1.085 ms = 1843 = 0733H where TH = 07H and TL = 33H
10. 100 ms/1.085 ms = 92 or 5CH; therefore, TH = 5CH

SECTION 9.2: COUNTER PROGRAMMING

1. The crystal attached to the 8051
2. The clock source for the timers comes from pins T0 and T1.
3. Yes
4. We must use the instruction "SETB P3.4" to configure the T1 pin as input, which allows the clocks to come from an external source. This is because all ports are configured as output upon reset.
5. SETB TR1

SECTION 9.3: PROGRAMMING TIMERS 0 AND 1 IN 8051 C

1. The crystal attached to the 8051
2. Timer 2, mode 2, 8-bit auto reload
3. FFFFH to 0
4. FFH to 0
5. 38H
6. TMOD
7. Yes
8. while (TF1==0);

CHAPTER 10

8051 SERIAL PORT PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

- **>> Contrast and compare serial versus parallel communication**
- **>> List the advantages of serial communication over parallel**
- **>> Explain serial communication protocol**
- **>> Contrast synchronous versus asynchronous communication**
- **>> Contrast half- versus full-duplex transmission**
- **>> Explain the process of data framing**
- **>> Desribe data transfer rate and bps rate**
- **>> Define the RS232 standard**
- **>> Explain the use of the MAX232 and MAX233 chips**
- **>> Interface the 8051 with an RS232 connector**
- **>> Discuss the baud rate of the 8051**
- **>> Describe serial communication features of the 8051**
- **>> Program the 8051 serial port in Assembly and C**
- **>> Program the second serial port of DS89C4x0 in Assembly and C**

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Examples of parallel transfers are printers and hard disks; each uses cables with many wire strips. Although in such cases a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. Serial communication of the 8051 is the topic of this chapter. The 8051 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

In this chapter we first discuss the basics of serial communication. In Section 10.2, 8051 interfacing to RS232 connectors via MAX232 line drivers is discussed. Serial port programming of the 8051 is discussed in Section 10.3. The second serial port of DS89C4x0 is programmed in Section 10.4. Section 10.5 covers 8051 C programming for serial ports #0 and #1.

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

When a microprocessor communicates with the outside world, it provides the data in byte-sized chunks. In some cases, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the printer. This can work only if the cable is not too long, since long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart. Figure 10-1 diagrams serial versus parallel data transfers.

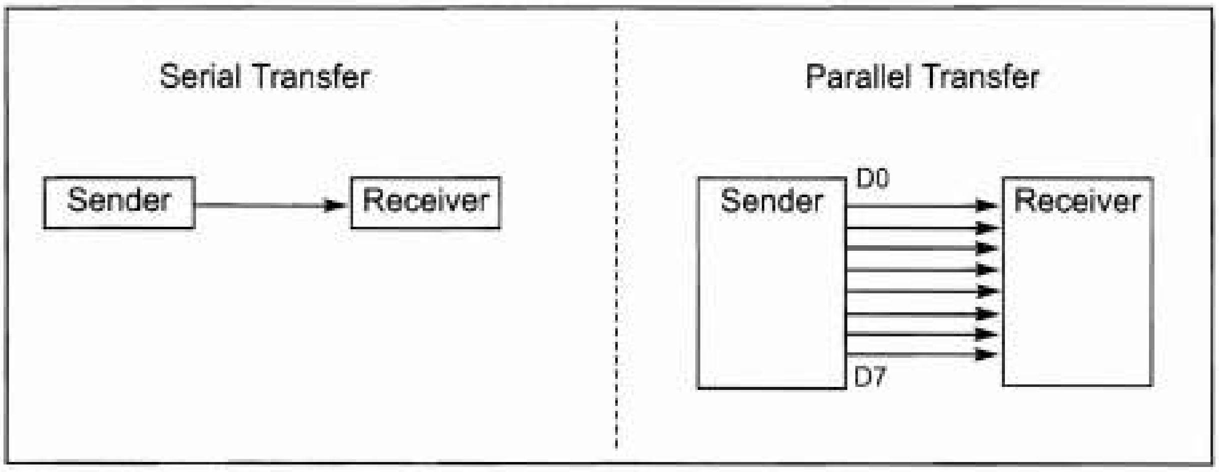


Figure 10-1. Serial versus Parallel Data Transfer

The fact that serial communication uses a single data line instead of the 8-bit data line of parallel communication not only makes it much cheaper but also enables two computers located in two different cities to communicate over the telephone.

For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte. Of course, if data is to be transferred on the telephone line, it must be converted from 0s and 1s to audio tones, which are sinusoidal-shaped signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how IBM PC keyboards transfer data to the motherboard. However, for long-distance data transfers using communication lines such as a telephone, serial data communication requires a modem to *modulate* (convert from 0s and 1s to audio tones) and *demodulate* (converting from audio tones to 0s and 1s).

Serial data communication uses two methods, asynchronous and synchronous. The *synchronous* method transfers a block of data (characters) at a time, while the *asynchronous* method transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, there are special IC chips made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The 8051 chip has a built-in UART, which is discussed in detail in Section 10.3.

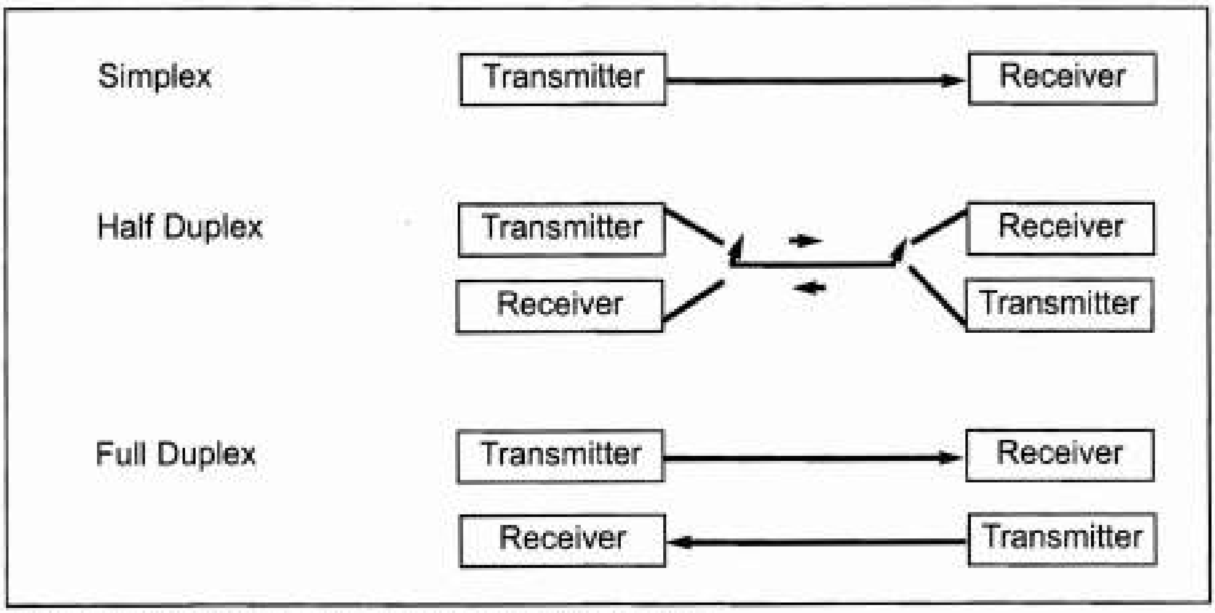


Figure 10-2. Simplex, Half-, and Full-Duplex Transfers

Half- and full-duplex transmission

In data transmission if the data can be transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex*. If the data can go both ways at the same time, it is *full duplex*. Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously. See Figure 10-2.

Asynchronous serial communication and data framing

The data coming in at the receiving end of the data line in a serial data transfer is all 0s and 1s; it is difficult to make sense of the data unless the sender and receiver agree on a set of rules, a *protocol*, on how the data is packed, how many bits constitute a character, and when the data begins and ends.

Start and stop bits

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing*. In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure 10-3 in which the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

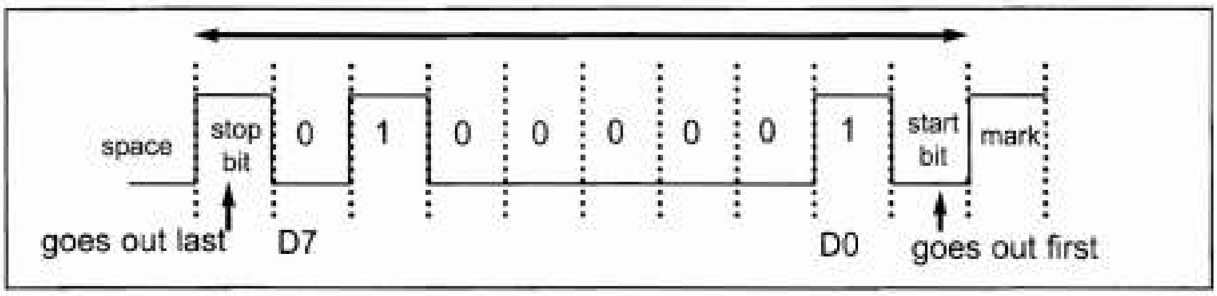


Figure 10-3. Framing ASCII "A" (41H)

Notice in Figure 10-3 that when there is no transfer, the signal is 1 (high), which is referred to as *mark*. The 0 (low) is referred to as *space*. Notice that the transmission begins with a start bit followed by D0, which is the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, due to the extended ASCII characters, 8-bit data has become common. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, for each 8-bit character there are an extra 2 bits, which gives 20% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd-parity bit the number of data bits, including the parity bit, has an odd number of 1s. Similarly, in an even-parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even-parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not necessarily equal. This is due to the fact that baud rate is the modem terminology and is defined as the number of signal changes per second. In modems a single change of signal, sometimes transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K bps. It must be noted that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. In this book we refer to it simply as RS232. Today, RS232 is the most widely used serial I/O interfacing standard. This standard is used in PCs and numerous types of equipment. However, since the standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by –3 to –25 V, while a 0 bit is +3 to +25 V, making -3 to +3 undefined. For this reason, to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage levels, and vice versa. MAX232 IC chips are commonly referred to as line drivers. RS232 connection to MAX232 is discussed in Section 10.2.

RS232 pins

Table 10-1 provides the pins and their labels for the RS232 cable, commonly referred to as the DB-25 connector. In labeling, DB-25P refers to the plug connector (male) and DB-25S is for the socket connector (female). See Figure 10-4.

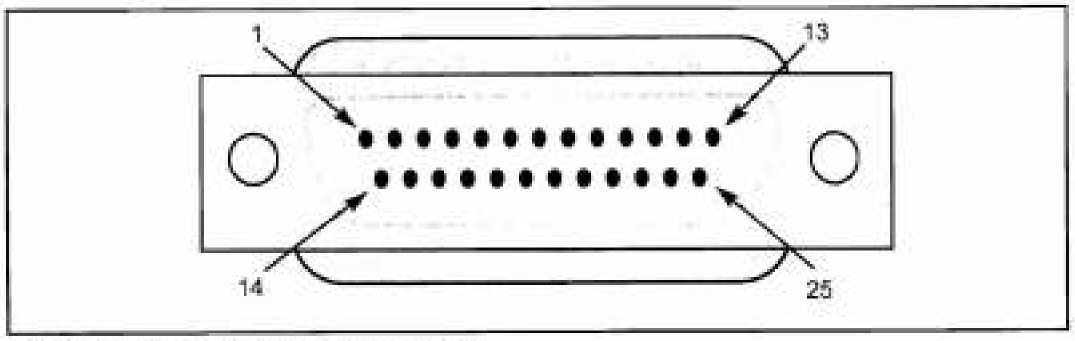


Figure 10-4. RS232 Connector DB-25

Since not all the pins are used in PC cables, IBM introduced the DB-9 version of the serial I/O standard, which uses 9 pins only, as shown in Table 10-2. The DB-9 pins are shown in Figure 10-5.

Data communication classification

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receive data, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data. Notice that all the RS232 pin function definitions of Tables 10-1 and 10-2 are from the DTE point of view.

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground, as shown in Figure 10-6. Notice in that figure that the RxD and TxD pins are interchanged.

Table 10-1: RS232 Pins (DB-25)

Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send ($\overline{\text{RTS}}$)
5	Clear to send ($\overline{\text{CTS}}$)
6	Data set ready ($\overline{\text{DSR}}$)
7	Signal ground (GND)
8	Data carrier detect ($\overline{\text{DCD}}$)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready ($\overline{\text{DTR}}$)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, because the receiving device in serial data communication may have no room for the data, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals. Their descriptions are provided below only as a reference and they can be bypassed since they are not supported by the 8051 UART chip.

1. DTR (data terminal ready). When a terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-low signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.
2. DSR (data set ready). When DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and input to the PC (DTE). This is an active-low signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.
3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-low output from the DTE and an input to the modem.
4. CTS (clear to send). In response to RTS, when the modem has room for storing the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.
5. DCD (carrier detect, or DCD, data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modem and an input to the PC (DTE).
6. RI (ring indicator). An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. It goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used, due to the fact that modems take care of answering the phone. However, if the PC is in charge of answering the phone, this signal can be used.

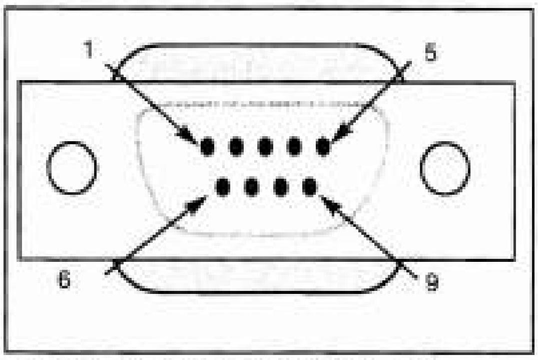


Figure 10-5. DB-9 9-Pin Connector

Table 10-2: IBM PC DB-9 Signals

Pin	Description
1	Data carrier detect ($\overline{DCD}$)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready ($\overline{DSR}$)
7	Request to send (RTS)
8	Clear to send ($\overline{CTS}$)
9	Ring indicator (RI)

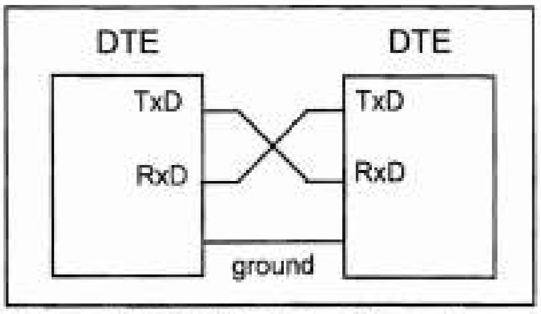


Figure 10-6. Null Modem Connection

From the above description, PC and modem communication can be summarized as follows: While signals DTR and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, if the modem is ready (has room) to accept the data, it sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

This concludes the description of the most important pins of the RS232 handshake signals plus TxD, RxD, and ground. Ground is also referred to as SG (signal ground).

IBM PC/compatible COM ports

IBM PC/compatible computers based on x86 (8086, 286, 386, 486, and Pentium) microprocessors normally have two COM ports. Both COM ports have RS232-type connectors. Many PCs use one each of the DB-25 and DB-9 RS232 connectors. The COM ports are designated as COM 1 and COM 2. At the present time COM 1 is used for the mouse and COM 2 is available for devices such as a modem. We can connect the 8051 serial port to the COM 2 port of a PC for serial communication experiments.

With this background in serial communication, we are ready to look at the 8051. In the next section we discuss the physical connection of the 8051 and RS232 connector, and in Section 10.3 we show how to program the 8051 serial communication port.

Review Questions

1. The transfer of data using parallel lines is ___________ (faster, slower) but ________________ (more expensive, less expensive).
2. True or false. Sending data to a printer is duplex.
3. True or false. In full duplex we must have two data lines, one for transfer and one for receive.
4. The start and stop bits are used in the ___________ (synchronous, asynchronous) method.
5. Assuming that we are transmitting the ASCII letter "E" (0100 0101 in binary) with no parity bit and one stop bit, show the sequence of bits transferred serially.
6. In Question 5, find the overhead due to framing.
7. Calculate the time it takes to transfer 10,000 characters as in Question 5 if we use 9600 bps. What percentage of time is wasted due to overhead?
8. True or false. RS232 is not TTL-compatible.
9. What voltage levels are used for binary 0 in RS232?
10. True or false. The 8051 has a built-in UART.
11. On the back of x86 PCs, we normally have ____ COM port connectors.
12. The PC COM ports are designated by DOS and Windows as ______ and ______.

SECTION 10.2: 8051 CONNECTION TO RS232

In this section, the details of the physical connections of the 8051 to RS232 connectors are given. As stated in Section 10.2, the RS232 standard is not TTL compatible; therefore, it requires a line driver such as the MAX232 chip to convert RS232 voltage levels to TTL levels, and vice versa. The interfacing of 8051 with RS232 connectors via the MAX232 chip is the main topic of this section.

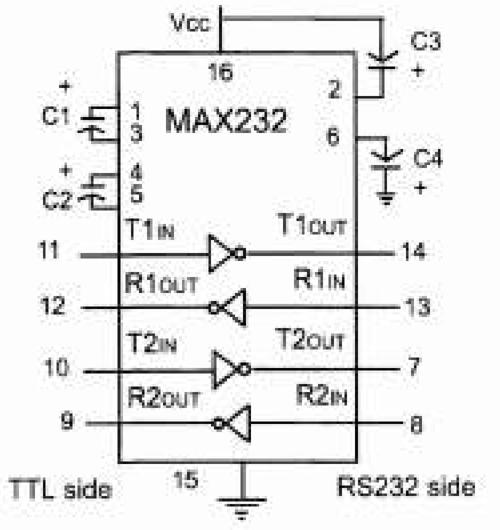
RxD and TxD pins in the 8051

The 8051 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TxD and RxD and are part of the port 3 group (P3.0 and P3.1). Pin 11 of the 8051 (P3.1) is assigned to TxD and pin 10 (P3.0) is designated as RxD. These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible. One such line driver is the MAX232 chip. This is discussed next.

MAX232

Since the RS232 is not compatible with today's microprocessors and microcontrollers, we need a line driver (voltage converter) to convert the RS232's signals to TTL voltage levels that will be acceptable to the 8051's TxD and RxD pins. One example of such a converter is MAX232 from Maxim Corp. (www.maxim-ic.com). The MAX232 converts from RS232 voltage levels to TTL voltage levels, and vice versa. One advantage of the MAX232 chip is that it uses a +5 V power source which, is the same as the source voltage for the 8051. In other words, with a single +5 V power supply we can power both the 8051 and MAX232, with no need for the dual power supplies that are common in many older systems.

The MAX232 has two sets of line drivers for transferring and receiving data, as shown in Figure 10-7. The line drivers used for TxD are called T1 and T2,



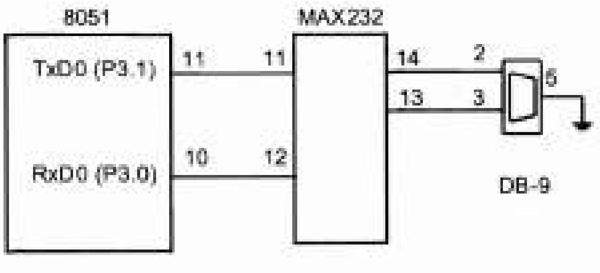


Figure 10-7. (a) Inside MAX232 and (b) its Connection to the 8051 (Null Modem)

while the line drivers for RxD are designated as R1 and R2. In many applications only one of each is used. For example, T1 and R1 are used together for TxD and RxD of the 8051, and the second set is left unused. Notice in MAX232 that the T1 line driver has a designation of T1in and T1out on pin numbers 11 and 14, respectively. The T1in pin is the TTL side and is connected to TxD of the microcontroller, while T1out is the RS232 side that is connected to the RxD pin of the RS232 DB connector. The R1 line driver has a designation of R1in and R1out on pin numbers 13 and 12, respectively. The R1in (pin 13) is the RS232 side that is connected to the TxD pin of the RS232 DB connector, and R1out (pin 12) is the TTL side that is connected to the RxD pin of the microcontroller. See Figure 10-7. Notice the null modem connection where RxD for one is TxD for the other.

MAX232 requires four capacitors ranging from 1 to 22 μF. The most widely used value for these capacitors is 22 μF.

MAX233

To save board space, some designers use the MAX233 chip from Maxim. The MAX233 performs the same job as the MAX232 but eliminates the need for capacitors. However, the MAX233 chip is much more expensive than the MAX232. Notice that MAX233 and MAX232 are not pin compatible. You cannot take a MAX232 out of a board and replace it with a MAX233. See Figure 10-8 for MAX233 with no capacitor used.

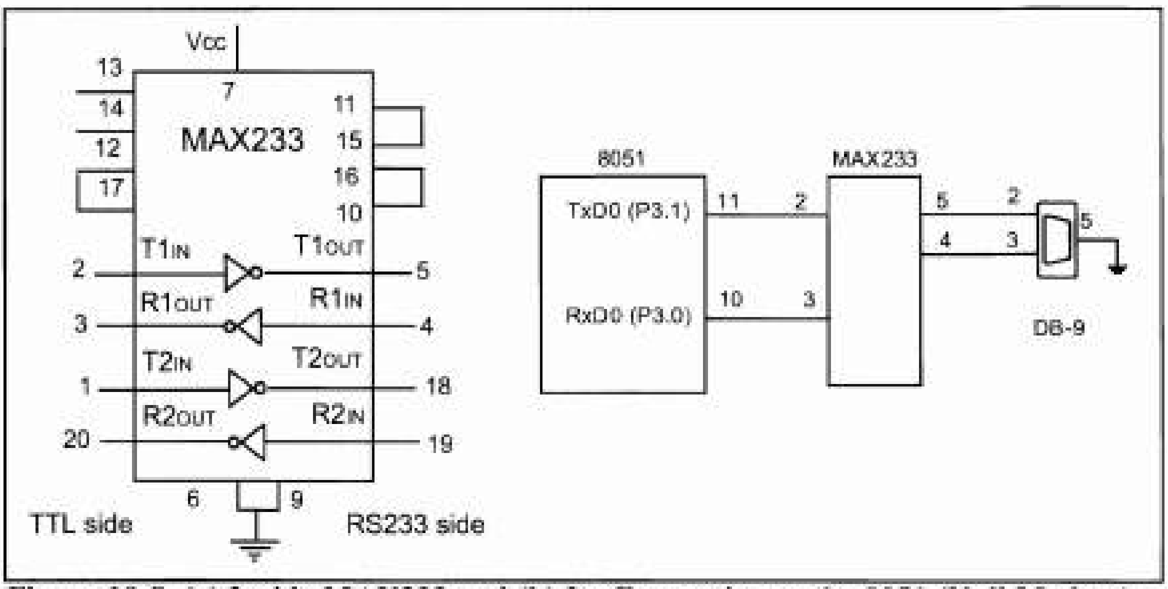


Figure 10-8. (a) Inside MAX233 and (b) Its Connection to the 8051 (Null Modem)

Review Questions

1. True or false. The PC COM port connector is the RS232 type.
2. Which pins of the 8051 are set aside for serial communication, and what are their functions?
3. What are line drivers such as MAX 232 used for?
4. MAX232 can support ____ lines for TxD and ____ lines for RxD.
5. What is the advantage of the MAX233 over the MAX232 chip?

SECTION 10.3: 8051 SERIAL PORT PROGRAMMING IN ASSEMBLY

Table 10-3: PC Baud Rates

110
150
300
600
1200
2400
4800
9600
19200

Note: Some of the Baud rates supported by 486/ Pentium IBM PC BIOS.

In this section we discuss the serial communication registers of the 8051 and show how to program them to transfer and receive data serially. Since IBM PC/compatible computers are so widely used to communicate with 8051-based systems, we will emphasize serial communications of the 8051 with the COM port of the PC. To allow data transfer between the PC and an 8051 system without any error, we must make sure that the baud rate of the 8051 system matches the baud rate of the PC's COM port. Some of the baud rates supported by PC BIOS are listed in Table 10-3. You can examine these baud rates by going to the Windows HyperTerminal program and clicking on the Communication Settings option. The HyperTerminal program comes with Windows. HyperTerminal supports baud rates much higher than the ones listed in Table 10-3.

Baud rate in the 8051

The 8051 transfers and receives data serially at many different baud rates. The baud rate in the 8051 is programmable. This is done with the help of Timer 1. Before we discuss how to do that, we will look at the relationship between the crystal frequency and the baud rate in the 8051.

As discussed in previous chapters, the 8051 divides the crystal frequency by 12 to get the machine cycle frequency. In the case of XTAL = 11.0592 MHz, the machine cycle frequency is 921.6 kHz (11.0592 MHz / 12 = 921.6 kHz). The 8051's serial communication UART circuitry divides the machine cycle frequency of 921.6 kHz by 32 once more before it is used by Timer 1 to set the baud rate. Therefore, 921.6 kHz divided by 32 gives 28,800 Hz. This is the number we will use throughout this section to find the Timer 1 value to set the baud rate. When Timer 1 is used to set the baud rate it must be programmed in mode 2, that is 8-bit, auto-reload. To get baud rates compatible with the PC, we must load TH1 with the values shown in Table 10-4. Example 10-1 shows how to verify the data in Table 10-4.

Table 10-4: Timer 1 TH1 Register Values for Various Baud Rates

Baud Rate	TH1 (Decimal)	TH1 (Hex)
9600	–3	FD
4800	-6	FA
2400	-12	F4
1200	-24	E8

Note: XTAL = 11.0592 MHz.

Example 10-1

With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 2400 (c) 1200

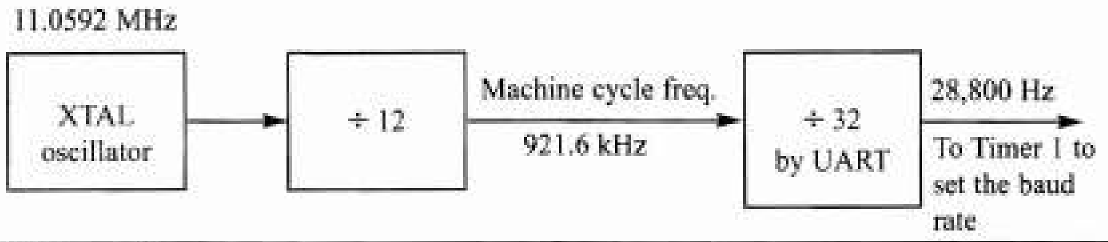
Solution:

With XTAL = 11.0592 MHz, we have:

The machine cycle frequency of the 8051 = 11.0592 MHz / 12 = 921.6 kHz, and 921.6 kHz / 32 = 28,800 Hz is the frequency provided by UART to Timer 1 to set baud rate.

(a) 28,800 / 3 = 9600	where –3 = FD (hex) is loaded into TH1
(b) 28,800 / 12 = 2400	where –12 = F4 (hex) is loaded into TH1
(c) 28,800 / 24 = 1200	where –24 = E8 (hex) is loaded into TH1

Notice that 1/12th of the crystal frequency divided by 32 is the default value upon activation of the 8051 RESET pin. We can change this default setting. This is explained at the end of this chapter.



SBUF register

SBUF is an 8-bit register used solely for serial communication in the 8051. For a byte of data to be transferred via the TxD line, it must be placed in the SBUF register. Similarly, SBUF holds the byte of data when it is received by the 8051's RxD line. SBUF can be accessed like any other register in the 8051. Look at the following examples of how this register is accessed:

```
MOV SBUF,#'D'        ;load SBUF=44H, ASCII for 'D'
MOV SBUF,A           ;copy accumulator into SBUF
MOV A,SBUF           ;copy SBUF into accumulator
```

The moment a byte is written into SBUF, it is framed with the start and stop bits and transferred serially via the TxD pin. Similarly, when the bits are received serially via RxD, the 8051 deframes it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the SBUF.

SCON (serial control) register

The SCON register is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other things.

The following describes various bits of the SCON register.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode specifier
SM1	SCON.6	Serial port mode specifier
SM2	SCON.5	Used for multiprocessor communication. (Make it 0.)
REN	SCON.4	Set/cleared by software to enable/disable reception.
TB8	SCON.3	Not widely used.
RB8	SCON.2	Not widely used.
TI	SCON.1	Transmit interrupt flag. Set by hardware at the beginning of the stop bit in mode 1. Must be cleared by software.
RI	SCON.0	Receive interrupt flag. Set by hardware halfway through the stop bit time in mode 1. Must be cleared by software.

Note: Make SM2, TB8, and RB8 = 0.

Figure 10-9. SCON Serial Port Control Register (Bit-Addressable)

SM0, SM1

SM0 and SM1 are D7 and D6 of the SCON register, respectively. These two bits determine the framing of data by specifying the number of bits per character, and the start and stop bits. They take the following combinations.

SM0	*SM1*	
0	0	Serial Mode 0
0	1	Serial Mode 1, 8-bit data, 1 stop bit, 1 start bit
1	0	Serial Mode 2
1	1	Serial Mode 3

Of the 4 serial modes, only mode 1 is of interest to us. Further explanation for the other three modes is in Appendix A.2. They are rarely used today. In the SCON register, when serial mode 1 is chosen, the data framing is 8 bits, 1 stop bit, and 1 start bit, which makes it compatible with the COM port of IBM/compatible PCs. More importantly, serial mode 1 allows the baud rate to be variable and is set by Timer 1 of the 8051. In serial mode 1, for each character a total of 10 bits are transferred, where the first bit is the start bit, followed by 8 bits of data, and finally 1 stop bit.

SM2

SM2 is the D5 bit of the SCON register. This bit enables the multiprocessing capability of the 8051 and is beyond the discussion of this chapter. For our applications, we will make SM2 = 0 since we are not using the 8051 in a multiprocessor environment.

REN

The REN (receive enable) bit is D4 of the SCON register. The REN bit is also referred to as SCON.4 since SCON is a bit-addressable register. When the REN bit is high, it allows the 8051 to receive data on the RxD pin of the 8051. As a result if we want the 8051 to both transfer and receive data, REN must be set to 1. By making REN = 0, the receiver is disabled. Making REN = 1 or REN = 0 can be achieved by the instructions "`SETB SCON.4`" and "`CLR SCON.4`", respectively. Notice that these instructions use the bit-addressable features of register SCON. This bit can be used to block any serial data reception and is an extremely important bit in the SCON register.

TB8

TB8 (transfer bit 8) is bit D3 of SCON. It is used for serial modes 2 and 3. We make TB8 = 0 since it is not used in our applications.

RB8

RB8 (receive bit 8) is bit D2 of the SCON register. In serial mode 1, this bit gets a copy of the stop bit when an 8-bit data is received. This bit (as is the case for TB8) is rarely used anymore. In all our applications we will make RB8 = 0. Like TB8, the RB8 bit is also used in serial modes 2 and 3.

TI

TI (transmit interrupt) is bit D1 of the SCON register. This is an extremely important flag bit in the SCON register. When the 8051 finishes the transfer of the 8-bit character, it raises the TI flag to indicate that it is ready to transfer another byte. The TI bit is raised at the beginning of the stop bit. We will discuss its role further when programming examples of data transmission are given.

RI

RI (receive interrupt) is the D0 bit of the SCON register. This is another extremely important flag bit in the SCON register. When the 8051 receives data serially via RxD, it gets rid of the start and stop bits and places the byte in the SBUF register. Then it raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost. RI is raised halfway through the stop bit, and we will soon see how this bit is used in programs for receiving data serially.

Programming the 8051 to transfer data serially

In programming the 8051 to transfer character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. The TH1 is loaded with one of the values in Table 10-4 to set the baud rate for serial data transfer (assuming XTAL = 11.0592 MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits.
4. TR1 is set to 1 to start Timer 1.
5. TI is cleared by the "`CLR TI`" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction "`JNB TI,xx`" to see if the character has been transferred completely.
8. To transfer the next character, go to Step 5.

Example 10-2 shows a program to transfer data serially at 4800 baud. Example 10-3 shows how to transfer "YES" continuously.

Example 10-2

Write a program for the 8051 to transfer letter "A" serially at 4800 baud, continuously.

Solution:

```
             MOV   TMOD,#20H  ;Timer 1, mode 2(auto-reload)
             MOV   TH1,#-6    ;4800 baud rate
             MOV   SCON,#50H  ;8-bit, 1 stop, REN enabled
             SETB  TR1        ;start Timer 1
AGAIN:       MOV   SBUF,#"A"  ;letter "A" to be transferred
HERE:        JNB   TI,HERE    ;wait for the last bit
             CLR   TI         ;clear TI for next char
             SJMP  AGAIN      ;keep sending A
```

Example 10-3

Write a program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
             MOV    TMOD,#20H ;Timer 1, mode 2
             MOV    TH1,#-3   ;9600 baud
             MOV    SCON,#50H ;8-bit, 1 stop bit, REN enabled
             SETB   TR1       ;start Timer 1
AGAIN:       MOV    A,#"Y"    ;transfer "Y"
             ACALL  TRANS
             MOV    A,#"E"    ;transfer "E"
             ACALL  TRANS
             MOV    A,#"S"    ;transfer "S"
             ACALL  TRANS
             SJMP   AGAIN     ;keep doing it
;-----serial data transfer subroutine
TRANS:       MOV    SBUF,A    ;load SBUF
HERE:        JNB    TI,HERE   ;wait for last bit to transfer
             CLR    TI        ;get ready for next byte
             RET
```

Importance of the TI flag

To understand the importance of the role of TI, look at the following sequence of steps that the 8051 goes through in transmitting a character via TxD.

1. The byte character to be transmitted is written into the SBUF register.
2. The start bit is transferred.
3. The 8-bit character is transferred one bit at a time.
4. The stop bit is transferred. It is during the transfer of the stop bit that the 8051 raises the TI flag (TI = 1), indicating that the last character was transmitted and it is ready to transfer the next character.
5. By monitoring the TI flag, we make sure that we are not overloading the SBUF register. If we write another byte into the SBUF register before TI is raised, the untransmitted portion of the previous byte will be lost. In other words, when

the 8051 finishes transferring a byte, it raises the TI flag to indicate it is ready for the next character.

6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by the "`CLR TI`" instruction in order for this new byte to be transferred.

From the above discussion we conclude that by checking the TI flag bit, we know whether or not the 8051 is ready to transfer another byte. More importantly, it must be noted that the TI flag bit is raised by the 8051 itself when it finishes the transfer of data, whereas it must be cleared by the programmer with an instruction such as "`CLR TI`". It also must be noted that if we write a byte into SBUF before the TI flag bit is raised, we risk the loss of a portion of the byte being transferred. The TI flag bit can be checked by the instruction "`JNB TI,...`" or we can use an interrupt, as we will see in Chapter 11. In Chapter 11 we will show how to use interrupts to transfer data serially, and avoid tying down the microcontroller with instructions such as "`JNB TI,xx`".

Programming the 8051 to receive data serially

In the programming of the 8051 to receive character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. TH1 is loaded with one of the values in Table 10-4 to set the baud rate (assuming XTAL = 11.0592 MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where 8-bit data is framed with start and stop bits and receive enable is turned on.
4. TR1 is set to 1 to start Timer 1.
5. RI is cleared with the "`CLR RI`" instruction.
6. The RI flag bit is monitored with the use of the instruction "`JNB RI,xx`" to see if an entire character has been received yet.
7. When RI is raised, SBUF has the byte. Its contents are moved into a safe place.
8. To receive the next character, go to Step 5.

Examples 10-4 and 10-5 shows the coding of the above steps.

Example 10-4

Program the 8051 to receive bytes of data serially, and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

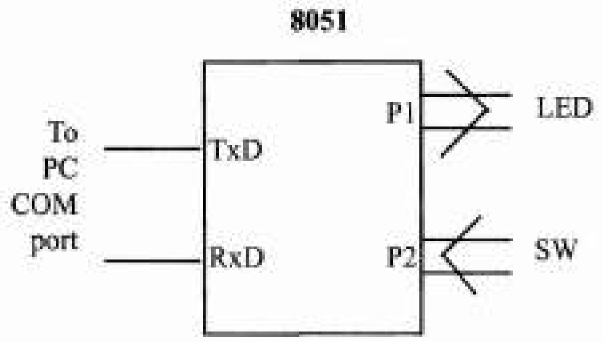
```
        MOV   TMOD,#20H  ;Timer 1, mode 2(auto-reload)
        MOV   TH1,#-6    ;4800 baud
        MOV   SCON,#50H  ;8-bit, 1 stop, REN enabled
        SETB  TR1        ;start Timer 1
HERE:   JNB   RI,HERE    ;wait for char to come in
        MOV   A,SBUF     ;save incoming byte in A
        MOV   P1,A       ;send to port 1
        CLR   RI         ;get ready to receive next byte
        SJMP  HERE       ;keep getting data
```

Example 10-5

Assume that the 8051 serial port is connected to the COM port of the IBM PC, and on the PC we are using the HyperTerminal program to send and receive data serially. P1 and P2 of the 8051 are connected to LEDs and switches, respectively. Write an 8051 program to (a) send to the PC the message "We Are Ready", (b) receive any data sent by the PC and put it on LEDs connected to P1, and (c) get data on switches connected to P2 and send it to the PC serially. The program should perform part (a) once, but parts (b) and (c) continuously. Use the 4800 baud rate.

Solution:

```
           ORG    0
           MOV    P2,#0FFH       ;make P2 an input port
           MOV    TMOD,#20H      ;Timer 1, mode 2(auto-reload)
           MOV    TH1,#0FAH      ;4800 baud rate
           MOV    SCON,#50H      ;8-bit,1 stop, REN enabled
           SETB   TR1            ;start Timer 1
           MOV    DPTR,#MYDATA   ;load pointer for message
H_1:       CLR    A
           MOVC   A,@A+DPTR      ;get the character
           JZ     B_1            ;if last character get out
           ACALL  SEND           ;otherwise call transfer
           INC    DPTR           ;next one
           SJMP   H_1            ;stay in loop
B_1:       MOV    A,P2           ;read data on P2
           ACALL  SEND           ;transfer it serially
           ACALL  RECV           ;get the serial data
           MOV    P1,A           ;display it on LEDs
           SJMP   B_1            ;stay in loop indefinitly
;--------------serial data transfer. ACC has the data
SEND:      MOV    SBUF,A         ;load the data
H_2:       JNB    TI,H_2         ;stay here until last bit gone
           CLR    TI             ;get ready for next char
           RET                   ;return to caller
;----------------receive data serially in ACC
RECV:      JNB    RI,RECV        ;wait here for char
           MOV    A,SBUF         ;save it in ACC
           CLR    RI             ;get ready for next char
           RET                   ;return to caller
;-----------------The message
MYDATA:    DB "We Are Ready",0
           END
```



Importance of the RI flag bit

In receiving bits via its RxD pin, the 8051 goes through the following steps.

1. It receives the start bit indicating that the next bit is the first bit of the character byte it is about to receive.
2. The 8-bit character is received one bit at time. When the last bit is received, a byte is formed and placed in SBUF.
3. The stop bit is received. When receiving the stop bit the 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character.
4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register. We copy the SBUF contents to a safe place in some other register or memory before it is lost.
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by the "`CLR RI`" instruction in order to allow the next received character byte to be placed in SBUF. Failure to do this causes loss of the received character.

From the above discussion we conclude that by checking the RI flag bit we know whether or not the 8051 has received a character byte. If we fail to copy SBUF into a safe place, we risk the loss of the received byte. More importantly, it must be noted that the RI flag bit is raised by the 8051, but it must be cleared by the programmer with an instruction such as "`CLR RI`". It also must be noted that if we copy SBUF into a safe place before the RI flag bit is raised, we risk copying garbage. The RI flag bit can be checked by the instruction "`JNB RI,xx`" or by using an interrupt, as we will see in Chapter 11.

Doubling the baud rate in the 8051

There are two ways to increase the baud rate of data transfer in the 8051.

1. Use a higher-frequency crystal.
2. Change a bit in the PCON register, shown below.

D7							D0
SMOD	--	--	--	GF1	GF0	PD	IDL

Option 1 is not feasible in many situations since the system crystal is fixed. More importantly, it is not feasible because the new crystal may not be compatible with the IBM PC serial COM port's baud rate. Therefore, we will explore option 2. There is a software way to double the baud rate of the 8051 while the crystal frequency is fixed. This is done with the register called PCON (power control). The PCON register is an 8-bit register. Of the 8 bits, some are unused, and some are used for the power control capability of the 8051. The bit that is used for the serial communication is D7, the SMOD (serial mode) bit. When the 8051 is powered up, D7 (SMOD bit) of the PCON register is zero. We can set it to high by software and thereby double the baud rate. The following sequence of instructions must be used to set high D7 of PCON, since it is not a bit-addressable register:

```
    MOV   A,PCON          ;place a copy of PCON in ACC
    SETB  ACC.7           ;make D7=1
    MOV   PCON,A          ;now SMOD=1 without
                          ;changing any other bits
```

To see how the baud rate is doubled with this method, we show the role of the SMOD bit (D7 bit of the PCON register), which can be 0 or 1. We discuss each case.

Baud rates for SMOD = 0

When SMOD = 0, the 8051 divides 1/12 of the crystal frequency by 32 and uses that frequency for Timer 1 to set the baud rate. In the case of XTAL = 11.0592 MHz we have:

```
Machine cycle freq. = 11.0592 MHz / 12 = 921.6 kHz
and
921.6 kHz / 32 = 28,800 Hz since SMOD = 0
```

This is the frequency used by Timer 1 to set the baud rate. This has been the basis of all the examples so far since it is the default when the 8051 is powered up. The baud rate for SMOD = 0 was listed in Table 10-4.

Baud rates for SMOD = 1

With the fixed crystal frequency, we can double the baud rate by making SMOD = 1. When the SMOD bit (D7 of the PCON register) is set to 1, 1/12 of XTAL is divided by 16 (instead of 32) and that is the frequency used by Timer 1 to set the baud rate. In the case of XTAL = 11.0592 MHz, we have:

```
Machine cycle freq. = 11.0592 MHz / 12 = 921.6 kHz
and
921.6 kHz / 16 = 57,600 Hz since SMOD = 1
```

This is the frequency used by Timer 1 to set the baud rate.

Table 10-5: Baud Rate Comparison for SMOD = 0 and SMOD = 1

TH1	(Decimal)	(Hex)	SMOD = 0	SMOD = 1
	-3	FD	9,600	19,200
	-6	FA	4,800	9,600
	-12	F4	2,400	4,800
	-24	E8	1,200	2,400

Note: XTAL = 11.0592 MHz.

Table 10-5 shows that the values loaded into TH1 are the same for both cases; however, the baud rates are doubled when SMOD = 1. Look at the following examples to clarify the data given in Table 10-5. See also Examples 10-6 through 10-10.

Example 10-6

Assuming that XTAL = 11.0592 MHz for the following program, state (a) what this program does, (b) compute the frequency used by Timer 1 to set the baud rate, and (c) find the baud rate of the data transfer.

```
      MOV   A,PCON       ;A = PCON
      SETB  ACC.7        ;make D7 = 1
      MOV   PCON,A       ;SMOD = 1, double baud rate
                         ;with same XTAL freq.
      MOV   TMOD,#20H    ;Timer 1, mode 2(auto-reload)
      MOV   TH1,-3       ;19200 (57,600 / 3 = 19200 baud rate
                         ;since SMOD=1)
      MOV   SCON,#50H    ;8-bit data,1 stop bit, RI enabled
      SETB  TR1          ;start Timer 1
      MOV   A,#"B"       ;transfer letter B
A_1:  CLR   TI           ;make sure TI=0
      MOV   SBUF,A       ;transfer it
H_1:  JNB   TI H_1       ;stay here until the last bit is gone
      SJMP  A_1          ;keep sending "B" again and again
```

Solution:

(a) This program transfers ASCII letter B (01000010 binary) continuously.
(b) With XTAL = 11.0592 MHz and SMOD = 1 in the above program, we have:

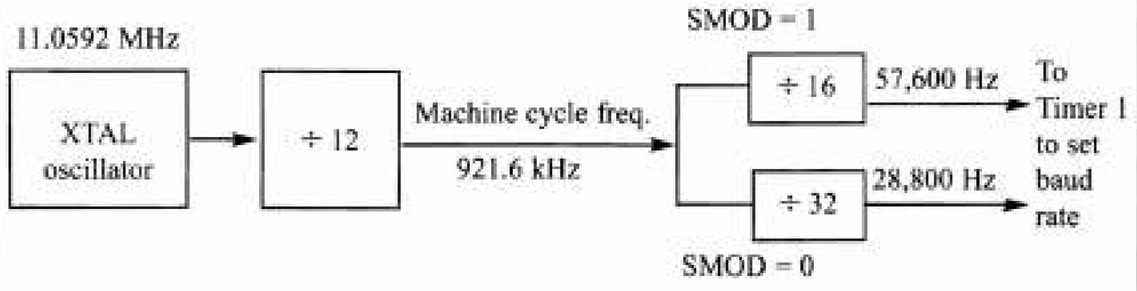
11.0592 MHz / 12 = 921.6 kHz machine cycle frequency
921.6 kHz / 16 = 57,600 Hz frequency used by Timer 1 to set the baud rate
57,600 Hz / 3 = 19,200 baud rate

Example 10-7

Find the TH1 value (in both decimal and hex) to set the baud rate to each of the following.
(a) 9600 (b) 4800 if SMOD = 1 Assume that XTAL = 11.0592 MHz.

Solution:

With XTAL = 11.0592 MHz and SMOD = 1, we have Timer 1 frequency = 57,600 Hz.
(a) 57,600 / 9600 = 6; therefore, TH1 = –6 or TH1 = FAH.
(b) 57,600 / 4800 = 12; therefore, TH1 = –12 or TH1 = F4H.



Example 10-8

Find the baud rate if TH1 = -2, SMOD = 1, and XTAL = 11.0592 MHz. Is this baud rate supported by IBM/compatible PCs?

Solution:

With XTAL = 11.0592 MHz and SMOD = 1, we have Timer 1 frequency = 57,600 Hz. The baud rate is 57,600 / 2 = 28,800. This baud rate is not supported by the BIOS of the PCs; however, the PC can be programmed to do data transfer at such a speed. Also, HyperTerminal in Windows supports this and other baud rates.

Example 10-9

Assume a switch is connected to pin P1.7. Write a program to monitor its status and send two messages to serial port continuously as follows:

SW=0 send "NO"

SW=1 send "YES"

Assume XTAL = 11.0592 MHz, 9600 baud, 8-bit data, and 1 stop bit.

Solution:

```
           SW1     EQU P1.7
           ORG     0H                 ;starting position
MAIN:      MOV     TMOD,#20H
           MOV     TH1,#-3            ;9600 baud rate
           MOV     SCON,#50H
           SETB    TR1                ;start timer
           SETB    SW1                ;make SW an input
S1:        JB      P1.7,NEXT          ;check SW status
           MOV     DPTR,#MESS1        ;if SW=0 display "NO"
FN:        CLR     A
           MOVC    A,@A+DPTR          ;read the value
           JZ      S1                 ;check for end of line
           ACALL   SENDCOM            ;send value to serial port
           INC     DPTR               ;move to next value
           SJMP    FN                 ;repeat
NEXT:      MOV     DPTR,#MESS2        ;if SW=1 display "YES"
LN:        CLR     A
           MOVC    A,@A+DPTR          ;read the value
           JZ      S1                 ;check for end of line
           ACALL   SENDCOM            ;send value to serial port
           INC     DPTR               ;move to next value
           SJMP    LN                 ;repeat
;------------------
SENDCOM:   MOV     SBUF,A             ;place value in buffer
HERE:      JNB     TI,HERE            ;wait until transmitted
           CLR     TI                 ;clear
           RET                        ;return
;------------------
MESS1:     DB      "NO",0
MESS2:     DB      "YES",0
                   END
```

Example 10-10

Write a program to send the message "The Earth is but One Country" to serial port. Assume a SW is connected to pin P1.2. Monitor its status and set the baud rate as follows:

SW = 0, 4800 baud rate
SW = 1, 9600 baud rate

Assume XTAL = 11.0592 MHz, 8-bit data, and 1 stop bit.

Solution:

```
          SW     BIT P1.2
          ORG    0H                ;Starting position
MAIN:
          MOV    TMOD,#20H
          MOV    TH1,#-6           ;4800 baud rate (default)
          MOV    SCON,#50H
          SETB   TR1
          SETB   SW                ;make SW an input
S1:       JNB    SW,SLOWSP         ;check SW status
          MOV    A,PCON            ;read PCON
          SETB   ACC.7             ;set SMOD High for 9600
          MOV    PCON,A            ;write PCON
          SJMP   OVER              ;send message
SLOWSP:   MOV    A,PCON            ;read PCON
          CLR    ACC.7             ;make SMOD Low for 4800
          MOV    PCON,A            ;write PCON
OVER:     MOV    DPTR,#MESS1       ;load address to message
FN:       CLR    A
          MOVC   A,@A+DPTR         ;read value
          JZ     S1                ;check for end of line
          ACALL  SENDCOM           ;send value to the serial port
          INC    DPTR              ;move to next value
          SJMP   FN                ;repeat

;-------------
SENDCOM:
          MOV    SBUF,A            ;place value in buffer
HERE:     JNB    TI,HERE           ;wait until transmitted
          CLR    TI                ;clear
          RET                      ;return

;------------------
MESS1:    DB    "The Earth is but One Country",0
          END
```

Interrupt-based data transfer

By now you might have noticed that it is a waste of the microcontroller's time to poll the TI and RI flags. In order to avoid wasting the microcontroller's time we use interrupts instead of polling. In Chapter 11, we will show how to use interrupts to program the 8051's serial communication port.

Review Questions

1. Which timer of the 8051 is used to set the baud rate?
2. If XTAL = 11.0592 MHz, what frequency is used by the timer to set the baud rate?
3. Which mode of the timer is used to set the baud rate?
4. With XTAL = 11.0592 MHz, what value should be loaded into TH1 to have a 9600 baud rate? Give the answer in both decimal and hex.
5. To transfer a byte of data serially, it must be placed in register _______.
6. SCON stands for __________ and it is a(n) ____-bit register.
7. Which register is used to set the data size and other framing information such as the stop bit?
8. True or false. SCON is a bit-addressable register.
9. When is TI raised?
10. Which register has the SMOD bit, and what is its status when the 8051 is powered up?

SECTION 10.4: PROGRAMMING THE SECOND SERIAL PORT

Many of the new generations of the 8051 microcontrollers come with two serial ports. The DS89C4x0 (DS89C420/30/40/...) and DS80C320 are among them. In this section we show the programming of the second serial port of the DS89C4x0 chip.

DS89C4x0 second serial port

The second serial port of the DS89C4x0 uses pins P1.2 and P1.3 for the Rx and Tx lines, respectively. See Figure 10-10. The MDE8051 Trainer (available from www.MicroDigitalEd.com) uses the DS89C4x0 chip and comes with two serial ports already installed. It also uses the MAX232 for the RS232 connection to DB9. The connections for the RS232 to the DS89C4x0 of the MDE8051 Trainer are shown in Figure 10-11. Notice that the first and second serial ports are designated as Serial #0 and Serial #1, respectively.

Addresses for all SCON and SBUF registers

All the programs we have seen so far in this chapter assume the use of the first serial port as the default serial port since every version of the 8051 comes with at least one serial port. The SCON, SBUF, and PCON registers of the 8051 are part of the special function registers. The address for each of the SFRs is shown in Table 10-6. Notice that SCON has address 98H, SBUF has address 99H, and finally PCON is assigned the 87H address. The first serial port is supported by all assemblers and C compilers in the market for the 8051. If you examine the list file for 8051 Assembly language programs, you will see that these labels are replaced with their SFR addresses. The second serial port is not implemented by all versions of the 8051/52 microcontroller. Only a few versions of the 8051/52, such as the DS89C4x0, come with the second serial port. As a result, the second serial port uses some reserved SFR addresses for the SCON and SBUF registers and there is no universal agreement among the makers as to which addresses should be used. In the case of the DS89C4x0, the SFR addresses of C0H and C1H are set aside for SBUF and SCON, as shown in Table 10-6. The DS89C4x0 technical documentation refers to these registers as SCON1 and SBUF1 since the first ones are designated as SCON0 and SBUF0.

Table 10-6: SFR Byte Addresses for DS89C4x0 Serial Ports

SFR	First Serial Port	Second Serial Port
SCON (byte address)	SCON0 = 98H	SCON1 = C0H
SBUF (byte address)	SBUF0 = 99H	SBUF1 = C1H
TL (byte address)	TL1 = 8BH	TL1 = 8BH
TH (byte address)	TH1 = 8DH	TH1 = 8DH
TCON (byte address)	TCON0 = 88H	TCON0 = 88H
PCON (byte address)	PCON = 87H	PCON = 87H

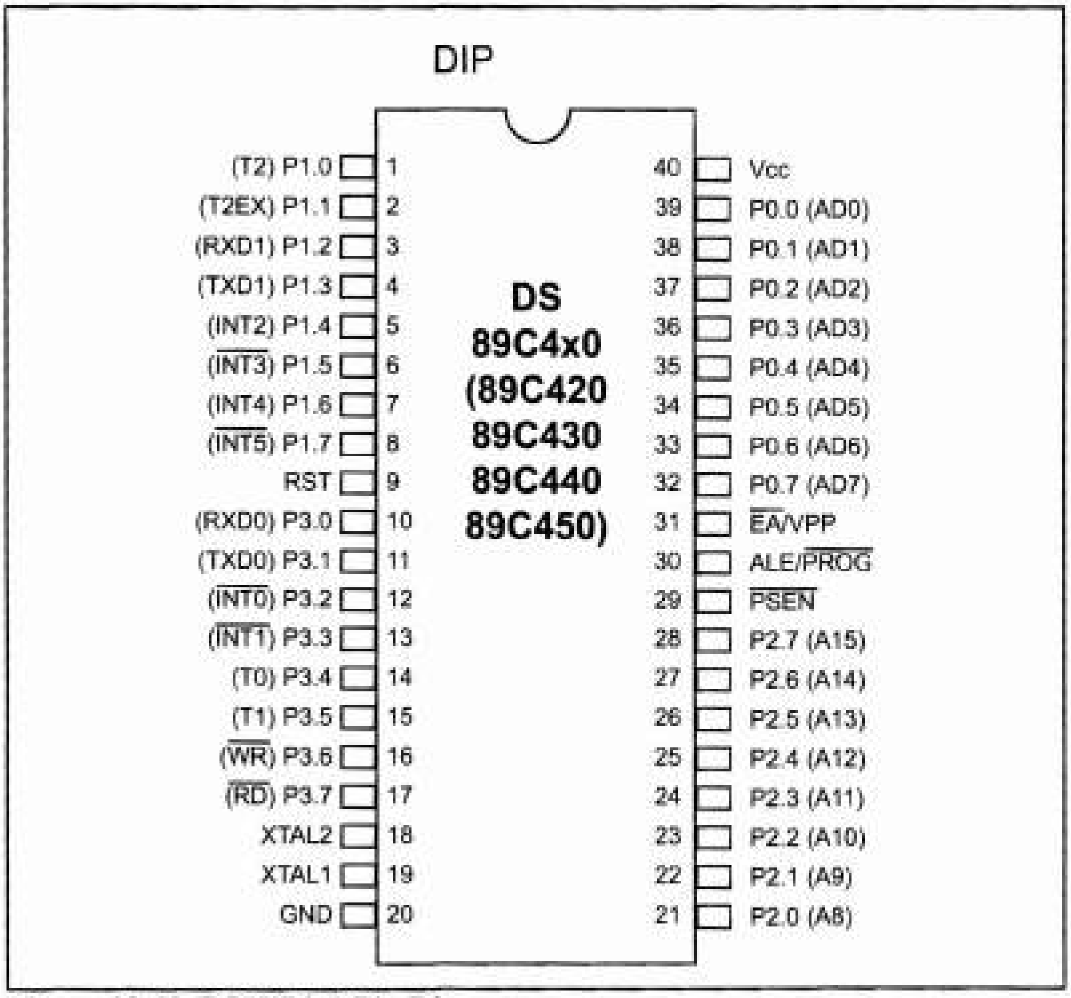


Figure 10-10. DS89C4x0 Pin Diagram

Note: Notice P1.2 and P1.3 pins are used by Rx and Tx lines of the 2nd serial port

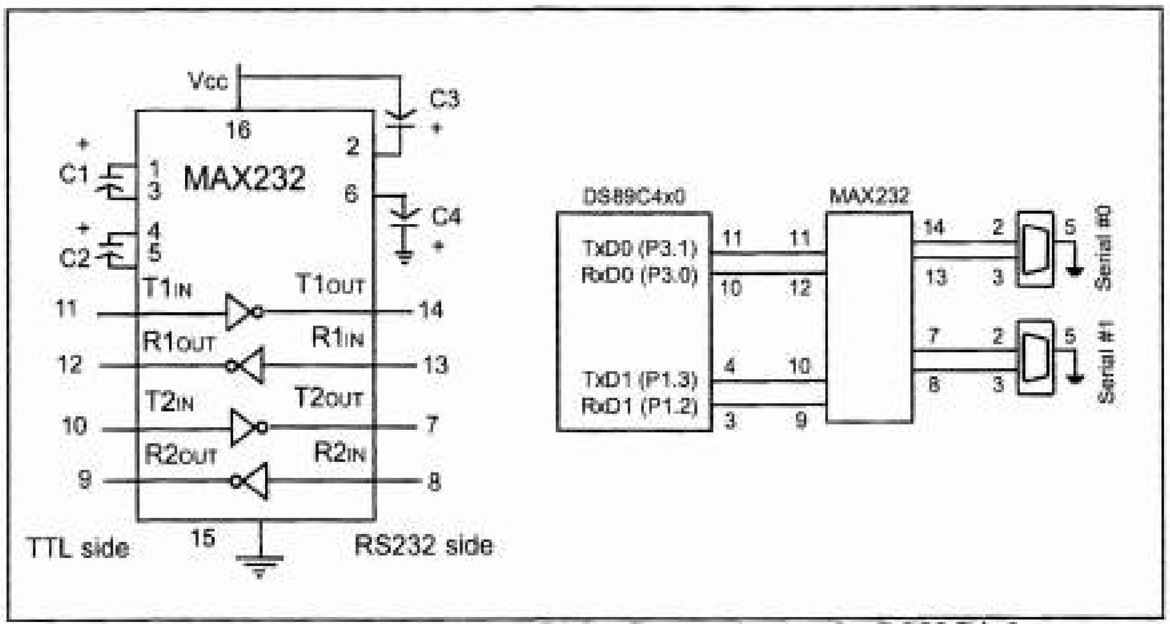


Figure 10-11. (a) Inside MAX232 and (b) its Connection to the DS89C4x0

Table 10-7: SFR Addresses for the DS89C4x0 (420, 430, etc.)

Symbol	Name	Address
ACC*	Accumulator	E0H
B*	B register	F0H
PSW*	Program status word	D0H
SP	Stack pointer	81H
DPTR	Data pointer 2 bytes	
DPL	Low byte	82H
DPH	High byte	83H
P0*	Port 0	80H
P1*	Port 1	90H
P2*	Port 2	0A0H
P3*	Port 3	B0H
IP*	Interrupt priority control	B8H
IE*	Interrupt enable control	A8H
TMOD	Timer/counter mode control	89H
TCON*	Timer/counter control	88H
T2CON*	Timer/counter 2 control	C8H
T2MOD	Timer/counter mode control	C9H
TH0	Timer/counter 0 high byte	8CH
TL0	Timer/counter 0 low byte	8AH
TH1	Timer/counter 1 high byte	8DH
TL1	Timer/counter 1 low byte	8BH
TH2	Timer/counter 2 high byte	CDH
TL2	Timer/counter 2 low byte	CCH
RCAP2H	T/C 2 capture register high byte	CBH
RCAP2L	T/C 2 capture register low byte	CAH
SCON0*	Serial control (first serial port)	98H
SBUF0	Serial data buffer (first serial port)	99H
PCON	Power control	87H
SCON1*	Serial control (second serial port)	C0H
SBUF1	Serial data buffer (second serial port)	C1H

* Bit-addressable

Programming the second serial port using timer 1

While each serial port has its own SCON and SBUF registers, both ports can use Timer 1 for setting the baud rate. Indeed, upon reset, the DS89C4x0 chip uses Timer 1 for setting the baud rate of both serial ports. Since the older 8051 assemblers do not support this new second serial port, we need to define them as shown in Example 10-11. Notice that in both C and Assembly, SBUF and SCON refer to the SFR registers of the first serial port. To avoid confusion, in DS89C4x0 programs we use SCON0 and SBUF0 for the first and SCON1 and SBUF1 for the second serial ports. For this reason, the MDE8051 Trainer designates the serial ports as Serial #0 and Serial #1 in order to comply with this designation.

See Examples 10-12 through 10-14.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Bits	Bit Addresses		
	Serial #0	Serial #1	
SM0	SCON0.7 = 9FH	SCON1.7 = C7H	Serial port mode specifier
SM1	SCON0.6 = 9EH	SCON1.6 = C6H	Serial port mode specifier
SM2	SCON0.5 = 9DH	SCON1.5 = C5H	Multiprocessor com.
REN	SCON0.4 = 9CH	SCON1.4 = C4H	Enable/disable reception
TB8	SCON0.3 = 9BH	SCON1.3 = C3H	Not widely used
RB8	SCON0.2 = 9AH	SCON1.2 = C2H	Not widely used
TI	SCON0.1 = 99H	SCON1.1 = C1H	Transmit interrupt flag
RI	SCON0.0 = 98H	SCON1.0 = C0H	Receive interrupt flag

Note: Make SM2, TB8, and RB8 = 0.

Figure 10-12. SCON0 and SCON1 Bit Addresses (TI and RI bits must be noted)

Example 10-11

Write a program for the second serial port of the DS89C4x0 to continuously transfer the letter "A" serially at 4800 baud. Use 8-bit data and 1 stop bit. Use Timer 1.

Solution:

```
           SBUF1  EQU 0C1H          ;second serial SBUF addr
           SCON1  EQU 0C0H          ;second serial SCON addr
           TI1    BIT 0C1H          ;second serial TI bit addr
           RI1    BIT 0C0H          ;second serial RI bit addr

           ORG    0H                ;starting position
MAIN:
           MOV    TMOD,#20H         ;COM2 uses Timer 1 upon reset
           MOV    TH1,#-6           ;4800 baud rate
           MOV    SCON1,#50H        ;COM2 has its own SCON1
           SETB   TR1               ;start Timer 1
AGAIN:     MOV    A,#'A'            ;send char 'A'
           ACALL  SENDCOM2
           SJMP   AGAIN
;-------------
SENDCOM2:
           MOV    SBUF1,A           ;COM2 has its own SBUF
HERE:      JNB    TI1,HERE          ;COM2 has its own TI flag
           CLR    TI1
           RET
           END
```

Example 10-12

Write a program to send the text string "Hello" to Serial #1. Set the baud rate at 9600, 8-bit data, and 1 stop bit.

Solution:

```
            SCON1  EQU 0C0H
            SBUF1  EQU 0C1H
            TI1    BIT 0C1H
            ORG    0H               ;starting position
            MOV    TMOD,#20H
            MOV    TH1,#-3          ;9600 baud rate
            MOV    SCON1,#50H
            SETB   TR1
            MOV    DPTR,#MESS1      ;display "Hello"
FN:         CLR    A
            MOVC   A,@A+DPTR        ;read value
            JZ     S1               ;check for end of line
            ACALL  SENDCOM2         ;send to serial port
            INC    DPTR             ;move to next value
            SJMP   FN
S1:         SJMP   S1
SENDCOM2:
            MOV    SBUF1,A          ;place value in buffer
HERE1:      JNB    TI1,HERE1        ;wait until transmitted
            CLR    TI1              ;clear
            RET
MESS1:      DB     "Hello",0
            END
```

Example 10-13

Program the second serial port of the DS89C4x0 to receive bytes of data serially and put them on P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
            SBUF1 EQU 0C1H    ;second serial SBUF addr
            SCON1 EQU 0C0H    ;second serial SCON addr
            RI1 BIT 0C0H      ;second serial RI bit addr
            ORG 0H            ;starting position
            MOV TMOD,#20H     ;COM2 uses Timer 1 upon reset
            MOV TH1,#-6       ;4800 baud rate
            MOV SCON1,#50H    ;COM2 has its own SCON1
            SETB TR1          ;start Timer 1
HERE:       JNB RI1,HERE      ;wait for data to come in
            MOV A,SBUF1       ;save data
            MOV P1,A          ;display on P1
            CLR RI1
            SJMP HERE
            END
```

Example 10-14

Assume that a switch is connected to pin P2.0.
Write a program to monitor the switch and perform the following:

(a) If SW = 0 send the message "Hello" to the Serial #0 port.
(b) If SW = 1 send the message "Goodbye" to the Serial #1 port.

Solution:

```
           SCON1  EQU 0C0H
           TI1    BIT 0C1H
           SW1    BIT P2.0
           ORG    0H              ;starting position
           MOV    TMOD,#20H
           MOV    TH1,#-3         ;9600 baud rate
           MOV    SCON,#50H
           MOV    SCON1,#50H
           SETB   TR1
           SETB   SW1             ;make SW1 an input
S1:        JB     SW1,NEXT        ;check SW1 status
           MOV    DPTR,#MESS1     ;if SW1=0 display "Hello"
FN:        CLR    A
           MOVC   A,@A+DPTR       ;read value
           JZ     S1              ;check for end of line
           ACALL  SENDCOM1        ;send to serial port
           INC    DPTR            ;move to next value
           SJMP   FN
NEXT:      MOV    DPTR,#MESS2     ;if SW1=1 display "Goodbye"
LN:        CLR    A
           MOVC   A,@A+DPTR       ;read value
           JZ     S1              ;check for end of line
           ACALL  SENDCOM2        ;send to serial port
           INC    DPTR            ;move to next value
           SJMP   LN

SENDCOM1:
           MOV    SBUF,A          ;place value in buffer
HERE:      JNB    TI,HERE         ;wait until transmitted
           CLR    TI              ;clear
           RET

SENDCOM2:
           MOV    SBUF1,A         ;place value in buffer
HERE1:     JNB    TI1,HERE1       ;wait until transmitted
           CLR    TI1             ;clear
           RET

MESS1:     DB     "Hello",0
MESS2:     DB     "Goodbye",0
           END
```

Review Questions

(All questions refer to the DS89C4x0 chip).

1. Upon reset, which timer is used to set the baud rate for Serial #0 and Serial #1?
2. Which pins are used for the second serial ports?
3. With XTAL = 11.0592 MHz, what value should be loaded into TH1 to have a 28,800 baud rate? Give the answer in both decimal and hex.
4. To transfer a byte of data via the second serial port, it must be placed in register _______.
5. SCON1 refers to _________ and it is a(n) ___-bit register.
6. Which register is used to set the data size and other framing information such as the stop bit for the second serial port?

SECTION 10.5: SERIAL PORT PROGRAMMING IN C

This section shows C programming of the serial ports for the 8051/52 and DS89C4x0 chips.

Transmitting and receiving data in 8051 C

As we stated in the last chapter, the SFR registers of the 8051 are accessible directly in 8051 C compilers by using the reg51.h file. Examples 10-15 through 10-19 show how to program the serial port in 8051 C. Connect your 8051 Trainer to the PC's COM port and use HyperTerminal to test the operation of these examples.

Example 10-15

Write a C program for the 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void)
  {
    TMOD=0x20;              //use Timer 1,8-BIT auto-reload
    TH1=0xFA;               //4800 baud rate
    SCON=0x50;
    TR1=1;
    while(1)
      {
        SBUF='A';           //place value in buffer
        while(TI==0);
        TI=0;
      }
  }
```

Example 10-16

Write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
#include <reg51.h>
void SerTx(unsigned char);
void main(void)
  {
    TMOD=0x20;          //use Timer 1,8-BIT auto-reload
    TH1=0xFD;           //9600 baud rate
    SCON=0x50;
    TR1=1;              //start timer
    while(1)
      {
        SerTx('Y');
        SerTx('E');
        SerTx('S');
      }
  }
void SerTx(unsigned char x)
  {
    SBUF=x;             //place value in buffer
    while(TI==0);       //wait until transmitted
    TI=0;
  }
```

Example 10-17

Program the 8051 in C to receive bytes of data serially and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
#include <reg51.h>
void main (void)
  {
    unsigned char mybyte;
    TMOD=0x20;                    //use Timer 1,8-BIT auto-reload
    TH1=0xFA;                     //4800 baud rate
    SCON=0x50;
    TR1=1;                        //start timer
    while(1)                      //repeat forever
      {
        while(RI==0);             //wait to receive
        mybyte=SBUF;              //save value
        P1=mybyte;                //write value to port
        RI=0;
      }
  }
```

Example 10-18

Write an 8051 C program to send two different strings to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and make a decision as follows:
SW = 0: send your first name
SW = 1: send your last name
Assume XTAL = 11.0592 MHz, baud rate of 9600, 8-bit data, 1 stop bit.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0;                     //input switch
void main(void)
  {
    unsigned char z;
    unsigned char fname[]="ALI";
    unsigned char lname[]="SMITH";
    TMOD=0x20;                      //use Timer 1,8-BIT auto-reload
    TH1=0xFD;                       //9600 baud rate
    SCON=0x50;
    TR1=1;                          //start timer
    if(MYSW==0)                     //check switch
      {
        for(z=0;z<3;z++)            //write name
          {
            SBUF=fname[z];          //place value in buffer
            while(TI==0);           //wait for transmit
            TI=0;
          }
      }
    else
      {
        for(z=0;z<5;z++)            //write name
          {
            SBUF=lname[z];          //place value in buffer
            while(TI==0);           //wait for transmit
            TI=0;
          }
      }
  }
```

Example 10-19

Write an 8051 C program to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and set the baud rate as follows:
SW = 0 28,800 baud rate
SW = 1 56K baud rate
Assume that XTAL = 11.0592 MHz for both cases.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0;                    //input switch
void main(void)
  {
    unsigned char z;
    unsigned char Mess1[]="Normal Speed";
    unsigned char Mess2[]="High Speed";
    TMOD=0x20;                     //use Timer 1,8-BIT auto-reload
    TH1=0xFF;                      //28,800 for normal speed
    SCON=0x50;
    TR1=1;                         //start timer
    if(MYSW==0)
      {
        for(z=0;z<12;z++)
          {
            SBUF=Mess1[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
    else
      {
        PCON=PCON|0x80;            //for high speed of 56K
        for(z=0;z<10;z++)
          {
            SBUF=Mess2[z];         //place value in buffer
            while(TI==0);          //wait for transmit
            TI=0;
          }
      }
  }
```

8051 C compilers and the second serial port

Since many C compilers do not support the second serial port of the DS89C4x0 chip, we have to declare the byte addresses of the new SFR registers using the sfr keyword. Table 10-6 and Figure 10-12 provide the SFR byte and bit addresses for the DS89C4x0 chip. Examples 10-20 and 10-21 show C versions of Examples 10-11 and 10-13 in Section 10.4.

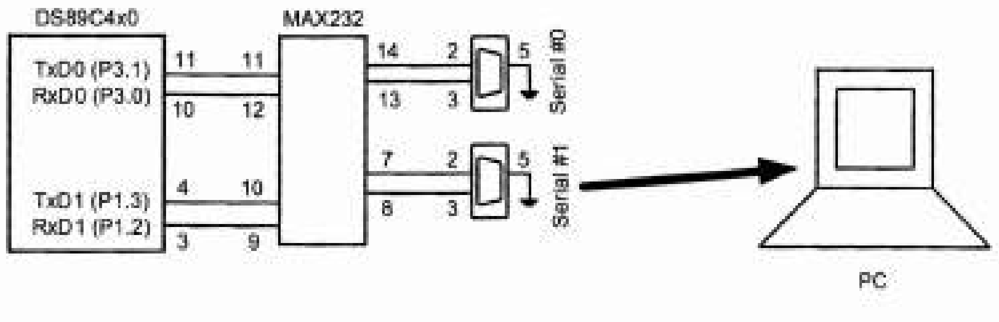
Notice in both Examples 10-20 and 10-21 that we are using Timer 1 to set the baud rate for the second serial port. Upon reset, Timer 1 is the default for the second serial port of the DS89C4x0 chip.

Example 10-20

Write a C program for the DS89C4x0 to transfer letter "A" serially at 4800 baud continuously. Use the second serial port with 8-bit data and 1 stop bit. We can only use Timer 1 to set the baud rate.

Solution:

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit TI1=0xC1;
void main(void)
  {
    TMOD=0x20;          //use Timer 1 for 2nd serial port
    TH1=0xFA;           //4800 baud rate
    SCON1=0x50;         //use 2nd serial port SCON1 register
    TR1=1;              //start timer
    while(1)
      {
        SBUF1='A';      //use 2nd serial port SBUF1 register
        while(TI1==0);  //wait for transmit
        TI1=0;
      }
  }
```



Example 10-21

Program the DS89C4x0 in C to receive bytes of data serially via the second serial port and put them in P1. Set the baud rate at 9600, 8-bit data, and 1 stop bit. Use Timer 1 for baud rate generation.

Solution:

```
#include <reg51.h>
sfr SBUF1=0xC1;
sfr SCON1=0xC0;
sbit RI1=0xC0;
void main(void)
  {
    unsigned char mybyte;
    TMOD=0x20;               //use Timer 1,8-BIT auto-reload
    TH1=0xFD;                //9600
    SCON1=0x50;              //use SCON1 of 2nd serial port
    TR1=1;
    while(1)
      {
        while(RI1==0);       //monitor RI1 of 2nd serial port
        mybyte=SBUF1;        //use SBUF1 of 2nd serial port
        P2=mybyte;           //place value on port
        RI1=0;
      }
  }
```

Review Questions

1. How are the SFR registers accessed in C?
2. True or false. C compilers support the second serial port of the DS89C420 chip.
3. Registers SBUF and SCON are declared in C using the ________ keyword.

See www.MicroDigitalEd.com on how to use Timer 2 to set the baud rate of Serial #0.

SUMMARY

This chapter began with an introduction to the fundamentals of serial communication. Serial communication, in which data is sent one bit a time, is used when data is sent over significant distances since in parallel communication, where data is sent a byte or more a time, great distances can cause distortion of the data. Serial communication has the additional advantage of allowing transmission over phone lines. Serial communication uses two methods: synchronous and asynchronous. In synchronous communication, data is sent in blocks of bytes; in asynchronous, data is sent in bytes. Data communication can be simplex (can send but cannot receive), half duplex (can send and receive, but not at the same time), or full duplex (can send and receive at the same time). RS232 is a standard for serial communication connectors.

The 8051's UART was discussed. We showed how to interface the 8051 with an RS232 connector and change the baud rate of the 8051. In addition, we described the serial communication features of the 8051, and programmed the 8051 for serial data communication. We also showed how to program the second serial port of the DS89C4x0 chip in Assembly and C.

PROBLEMS

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

1. Which is more expensive, parallel or serial data transfer?
2. True or false. 0- and 5-V digital pulses can be transferred on the telephone without being converted (modulated).
3. Show the framing of the letter ASCII "Z" (0101 1010), no parity, 1 stop bit.
4. If there is no data transfer and the line is high, it is called __________ (mark, space).
5. True or false. The stop bit can be 1, 2, or none at all.
6. Calculate the overhead percentage if the data size is 7, 1 stop bit, no parity.
7. True or false. The RS232 voltage specification is TTL compatible.
8. What is the function of the MAX 232 chip?
9. True or false. DB-25 and DB-9 are pin compatible for the first 9 pins.
10. How many pins of the RS232 are used by the IBM serial cable, and why?
11. True or false. The longer the cable, the higher the data transfer baud rate.
12. State the absolute minimum number of signals needed to transfer data between two PCs connected serially. What are those signals?
13. If two PCs are connected through the RS232 without the modem, they are both configured as a _______ (DTE, DCE) -to- ______ (DTE, DCE) connection.
14. State the nine most important signals of the RS232.
15. Calculate the total number of bits transferred if 200 pages of ASCII data are sent using asynchronous serial data transfer. Assume a data size of 8 bits, 1 stop bit, and no parity. Assume each page has 80x25 of text characters.
16. In Problem 15, how long will the data transfer take if the baud rate is 9,600?

SECTION 10.2: 8051 CONNECTION TO RS232

17. The MAX232 DIP package has _____ pins.
18. For the MAX232, indicate the V_{CC} and GND pins.
19. The MAX233 DIP package has ____ pins.
20. For the MAX233, indicate the V_{CC} and GND pins.
21. Is the MAX232 pin compatible with the MAX233?
22. State the advantages and disadvantages of the MAX232 and MAX233.

23. MAX232/233 has ____ line driver(s) for the RxD wire.
24. MAX232/233 has ____ line driver(s) for the TxD wire.
25. Show the connection of pins TxD and RxD of the 8051 to a DB-9 RS232 connector via the second set of line drivers of MAX232.
26. Show the connection of the TxD and RxD pins of the 8051 to a DB-9 RS232 connector via the second set of line drivers of MAX233.
27. Show the connection of the TxD and RxD pins of the 8051 to a DB-25 RS232 connector via MAX232.
28. Show the connection of the TxD and RxD pins of the 8051 to a DB-25 RS232 connector via MAX233.

SECTION 10.3: 8051 SERIAL PORT PROGRAMMING IN ASSEMBLY

29. Which of the following baud rates are supported by the BIOS of 486/Pentium PCs?

(a) 4,800	(b) 3,600	(c) 9,600
(d) 1,800	(e) 1,200	(f) 19,200

30. Which timer of the 8051 is used for baud rate programming?
31. Which mode of the timer is used for baud rate programming?
32. What is the role of the SBUF register in serial data transfer?
33. SBUF is a(n) ____-bit register.
34. What is the role of the SCON register in serial data transfer?
35. SCON is a(n) ____-bit register.
36. For XTAL = 11.0592 MHz, find the TH1 value (in both decimal and hex) for each of the following baud rates.
    (a) 9,600 (b) 4,800 (c) 1,200 (d) 300 (e) 150
37. What is the baud rate if we use "`MOV TH1,#-1`" to program the baud rate?
38. Write an 8051 program to transfer serially the letter "Z" continuously at a 1,200 baud rate.
39. Write an 8051 program to transfer serially the message "The earth is but one country and mankind its citizens" continuously at a 57,600 baud rate.
40. When is the TI flag bit raised?
41. When is the RI flag bit raised?
42. To which register do RI and TI belong? Is that register bit-addressable?
43. What is the role of the REN bit in the SCON register?
44. In a given situation we cannot accept reception of any serial data. How do you block such a reception with a single instruction?
45. To which register does the SMOD bit belong? State its role in the rate of data transfer.
46. Is the SMOD bit high or low when the 8051 is powered up?

In the following questions the baud rates are not compatible with the COM ports of the PC (x86 IBM/compatible).

47. Find the baud rate for the following if XTAL = 16 MHz and SMOD = 0.

(a) `MOV TH1,#-10`	(b) `MOV TH1,#-25`
(c) `MOV TH1,#-200`	(d) `MOV TH1,#-180`

48. Find the baud rate for the following if XTAL = 24 MHz and SMOD = 0.
   (a) `MOV TH1,#-15`   (b) `MOV TH1,#-24`
   (c) `MOV TH1,#-100`   (d) `MOV TH1,#-150`
49. Find the baud rate for the following if XTAL = 16 MHz and SMOD = 1.
   (a) `MOV TH1,#-10`   (b) `MOV TH1,#-25`
   (c) `MOV TH1,#-200`   (d) `MOV TH1,#-180`
50. Find the baud rate for the following if XTAL = 24 MHz and SMOD = 1.
   (a) `MOV TH1,#-15`   (b) `MOV TH1,#-24`
   (c) `MOV TH1,#-100`   (d) `MOV TH1,#-150`

SECTION 10.4: PROGRAMMING THE SECOND SERIAL PORT

51. Upon reset, which timer of the 8051 is used?
52. Which timer of the DS89C4x0 is used to set the baud rate for the second serial port?
53. Which mode of the timer is used for baud rate programming of the second serial port?
54. What is the role of the SBUF1 register in serial data transfer?
55. SBUF1 is a(n) ____-bit register.
56. What is the role of the SCON1 register in serial data transfer?
57. SCON1 is a(n) ____-bit register.
58. For XTAL = 11.0592 MHz, find the TH1 value (in both decimal and hex) for each of the following baud rates.
   (a) 9,600   (b) 4,800   (c) 1,200   (d) 300   (e) 150
59. Write a program for DS89C4x0 to transfer serially the letter "Z" continuously at a 1,200 baud rate. Use the second serial port.
60. Write a program for DS89C4x0 to transfer serially the message "The earth is but one country and mankind its citizens" continuously at a 57,600 baud rate. Use the second serial port.
61. When is the TI1 flag bit raised?

SECTION 10.5: SERIAL PORT PROGRAMMING IN C

62. Write an 8051 C program to transfer serially the letter "Z" continuously at a 1,200 baud rate.
63. Write an 8051 C program to transfer serially the message "The earth is but one country and mankind its citizens" continuously at a 57,600 baud rate.
64. Write a C program for DS89C4z0 to transfer serially the letter "Z" continuously at a 1,200 baud rate. Use the second serial port.
65. Write a C program for the DS89C4x0 to transfer serially the message "The earth is but one country and mankind its citizens" continuously at a 57,600 baud rate. Use the second serial port.

ANSWERS TO REVIEW QUESTIONS

SECTION 10.1: BASICS OF SERIAL COMMUNICATION

1. Faster, more expensive
2. False; it is simplex.
3. True
4. Asynchronous
5. With 0100 0101 binary the bits are transmitted in the sequence:
   (a) 0 (start bit) (b) 1 (c) 0 (d) 1 (e) 0 (f) 0 (g) 0 (h) 1 (i) 0 (j) 1 (stop bit)
6. 2 bits (one for the start bit and one for the stop bit). Therefore, for each 8-bit character, a total of 10 bits is transferred.
7. $10000 \times 10 = 100000$ bits total bits transmitted. $100000 / 9600 = 10.4$ seconds; $2 / 10 = 20\%$.
8. True
9. +3 to +25 V
10. True
11. 2
12. COM 1 and COM 2

SECTION 10.2: 8051 CONNECTION TO RS232

1. True
2. Pins 10 and 11. Pin 10 is for TxD and pin 11 for RxD.
3. They are used for converting from RS232 voltage levels to TTL voltage levels and vice versa.
4. 2, 2
5. It does not need the four capacitors that MAX232 must have.

SECTION 10.3: 8051 SERIAL PORT PROGRAMMING IN ASSEMBLY

1. Timer 1
2. 28,800 Hz
3. Mode 2
4. –3 or FDH since 28,800 / 3 = 9,600
5. SBUF
6. Serial control, 8
7. SCON
8. False
9. During transfer of stop bit
10. PCON; it is low upon RESET.

SECTION 10.4: PROGRAMMING THE SECOND SERIAL PORT

1. Timer 1
2. Pins P1.2 and P1.3
3. –1 of FFH
4. SBUF1
5. Serial Control 1, 8
6. SCON1

SECTION 10.5: SERIAL PORT PROGRAMMING IN C

1. By using the reg51.h file
2. False
3. sfr

CHAPTER 11

INTERRUPTS PROGRAMMING IN ASSEMBLY AND C

OBJECTIVES

Upon completion of this chapter, you will be able to:

>> Contrast and compare interrupts versus polling
>> Explain the purpose of the ISR (interrupt service routine)
>> List the 6 interrupts of the 8051
>> Explain the purpose of the interrupt vector table
>> Enable or disable 8051/52 interrupts
>> Program the 8051/52 timers using interrupts
>> Describe the external hardware interrupts of the 8051/52
>> Contrast edge-triggered with level-triggered interrupts
>> Program the 8051 for interrupt-based serial communication
>> Define the interrupt priority of the 8051
>> Program 8051/52 interrupts in C

In this chapter we explore the concept of the interrupt and interrupt programming. In Section 11.1 the basics of 8051 interrupts are discussed. In Section 11.2 interrupts belonging to Timers 0 and 1 are discussed. External hardware interrupts are discussed in Section 11.3, while the interrupt related to serial communication is presented in Section 11.4. In Section 11.5, we cover interrupt priority in the 8051/52. Finally, C programming of 8051 interrupts is covered in Section 11.6.

SECTION 11.1: 8051 INTERRUPTS

In this section, first we examine the difference between polling and interrupts and then describe the various interrupts of the 8051.

Interrupts vs. polling

A single microcontroller can serve several devices. There are two ways to do that: interrupts or polling. In the *interrupt* method, whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device. The program associated with the interrupt is called the *interrupt service routine* (ISR) or *interrupt handler*. In *polling*, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller. The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it. The polling method cannot assign priority since it checks all devices in a round-robin fashion. More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This is again not possible with the polling method. The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service. So in order to avoid tying down the microcontroller, interrupts are used. For example, in discussing timers in Chapter 9 we used the instruction "`JNB TF,target`", and waited until the timer rolled over, and while we were waiting we could not do anything else. That is a waste of the microcontroller's time that could have been used to perform some useful tasks. In the case of the timer, if we use the interrupt method, the microcontroller can go about doing other tasks, and when the TF flag is raised the timer will interrupt the microcontroller in whatever it is doing.

Interrupt service routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table, shown in Table 11-1.

Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps.

1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack.
2. It also saves the current status of all the interrupts internally (i.e., not on the stack).
3. It jumps to a fixed location in memory called the interrupt vector table that holds the address of the interrupt service routine.
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC. Then it starts to execute from that address.

Notice from Step 5 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal.

Six interrupts in the 8051

In reality, only five interrupts are available to the user in the 8051, but many manufacturers' data sheets state that there are six interrupts since they include reset. The six interrupts in the 8051 are allocated as follows.

1. Reset. When the reset pin is activated, the 8051 jumps to address location 0000. This is the power-up reset discussed in Chapter 4.
2. Two interrupts are set aside for the timers: one for Timer 0 and one for Timer 1. Memory locations 000BH and 001BH in the interrupt vector table belong to Timer 0 and Timer 1, respectively.
3. Two interrupts are set aside for hardware external hardware interrupts. Pin numbers 12 (P3.2) and 13 (P3.3) in port 3 are for the external hardware interrupts INT0 and INT1, respectively. These external interrupts are also referred to as EX1 and EX2. Memory locations 0003H and 0013H in the interrupt vector table are assigned to INT0 and INT1, respectively.
4. Serial communication has a single interrupt that belongs to both receive and transmit. The interrupt vector table location 0023H belongs to this interrupt.

Notice in Table 11-1 that a limited number of bytes is set aside for each interrupt. For example, a total of 8 bytes from location 0003 to 0000A is set aside for INT0, external hardware interrupt 0. Similarly, a total of 8 bytes from location 000BH to 0012H is reserved for TF0, Timer 0 interrupt. If the service routine for a given interrupt is short enough to fit in the memory space allocated to it, it is placed in the vector table; otherwise, an LJMP instruction is placed in the vector table to point to the address of the ISR. In that case, the rest of the bytes allocated to that interrupt are unused. In the next three sections we will see many examples of interrupt programming that clarify these concepts.

From Table 11-1, also notice that only three bytes of ROM space are assigned to the reset pin. They are ROM address locations 0, 1, and 2. Address location 3 belongs to external hardware interrupt 0. For this reason, in our program we put the LJMP as the first instruction and redirect the processor away from the interrupt vector table, as shown in Figure 11-1. In the next section we will see how this works in the context of some examples.

Table 11-1: Interrupt Vector Table for the 8051

Interrupt	ROM Location (Hex)	Pin	Flag Clearing
Reset	0000	9	Auto
External hardware interrupt 0 (INT0)	0003	P3.2 (12)	Auto
Timer 0 interrupt (TF0)	000B		Auto
External hardware interrupt 1 (INT1)	0013	P3.3 (13)	Auto
Timer 1 interrupt (TF1)	001B		Auto
Serial COM interrupt (RI and TI)	0023		Programmer clears it.

```
          ORG  0    ;wake-up ROM reset location
          LJMP MAIN ;bypass interrupt vector table

;---- the wake-up program
          ORG  30H
MAIN:
          ....
          END
```

Figure 11-1. Redirecting the 8051 from the Interrupt Vector Table at Power-up

Enabling and disabling an interrupt

Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them. There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts. Figure 11-2 shows the IE register. Note that IE is a bit-addressable register.

From Figure 11-2 notice that bit D7 in the IE register is called EA (enable all). This must be set to 1 in order for the rest of the register to take effect. D6 is unused. D5 is used by the 8052. The D4 bit is for the serial interrupt, and so on.

Steps in enabling an interrupt

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect.
2. If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high.

To understand this important point look at Example 11-1.

D7							D0
EA	--	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables Timer 1 overflow interrupt.
EX1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

Figure 11-2. IE (Interrupt Enable) Register

Example 11-1

Show the instructions to (a) enable the serial interrupt, Timer 0 interrupt, and external hardware interrupt 1 (EX1), and (b) disable (mask) the Timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

Solution:

```
(a)    MOV IE,#10010110B    ;enable serial, Timer 0, EX1
```

Since IE is a bit-addressable register, we can use the following instructions to access individual bits of the register.

```
(b)    CLR IE.1        ;mask(disable) Timer 0 interrupt only
(c)    CLR IE.7        ;disable all interrupts
```

Another way to perform the "`MOV IE,#10010110B`" instruction is by using single-bit instructions as shown below.

```
SETB IE.7      ;EA=1, Global enable
SETB IE.4      ;enable serial interrupt
SETB IE.1      ;enable Timer 0 interrupt
SETB IE.2      ;enable EX1
```

Review Questions

1. Of the interrupt and polling methods, which one avoids tying down the microcontroller?
2. Besides reset, how many interrupts do we have in the 8051?
3. In the 8051, what memory area is assigned to the interrupt vector table? Can the programmer change the memory space assigned to the table?
4. What are the contents of register IE upon reset, and what do these contents mean?
5. Show the instruction to enable the EX0 and Timer 0 interrupts.
6. Which pin of the 8051 is assigned to the external hardware interrupt INT1?
7. What address in the interrupt vector table is assigned to the INT1 and Timer 1 interrupts?

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

In Chapter 9 we discussed how to use Timer 0 and Timer 1 with the polling method. In this section we use interrupts to program the 8051 timers. Please review Chapter 9 before you study this section.

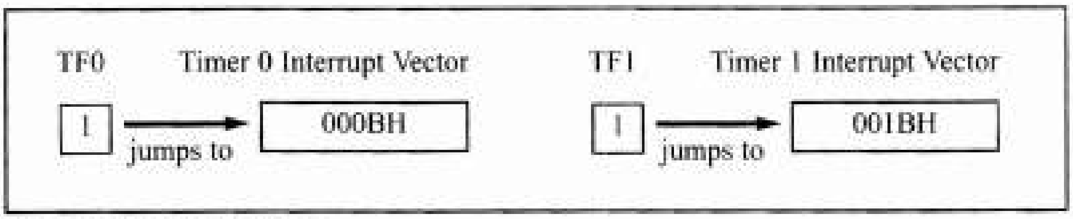


Figure 11-3. TF Interrupt

Roll-over timer flag and interrupt

In Chapter 9 we stated that the timer flag (TF) is raised when the timer rolls over. In that chapter, we also showed how to monitor TF with the instruction "`JNB TF, target`". In polling TF, we have to wait until the TF is raised. The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and cannot do any thing else. Using interrupts solves this problem and avoids tying down the controller. If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other things until it is notified that the timer has rolled over. See Figure 11-3 and Example 11-2.

Notice the following points about the program in Example 11-2.

1. We must avoid using the memory space allocated to the interrupt vector table. Therefore, we place all the initialization codes in memory starting at 30H. The LJMP instruction is the first instruction that the 8051 executes when it is powered up. LJMP redirects the controller away from the interrupt vector table.

2. The ISR for Timer 0 is located starting at memory location 000BH since it is small enough to fit the address space allocated to this interrupt.
3. We enabled the Timer 0 interrupt with "`MOV IE,#10000010B`" in MAIN.
4. While the P0 data is brought in and issued to P1 continuously, whenever Timer 0 is rolled over, the TF0 flag is raised, and the microcontroller gets out of the "BACK" loop and goes to 0000BH to execute the ISR associated with Timer 0.
5. In the ISR for Timer 0, notice that there is no need for a "`CLR TF0`" instruction before the RETI instruction. This is because the 8051 clears the TF flag internally upon jumping to the interrupt vector table.

Example 11-2

Write a program that continuously gets 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 μs period on pin P2.1. Use Timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use Timer 0 in mode 2 (auto-reload). TH0 = 100/1.085 μs = 92.

```
;--Upon wake-up go to main, avoid using memory space ;allocat-
ed to Interrupt Vector Table
           ORG   0000H
           LJMP  MAIN         ;bypass interrupt vector table
;
;--ISR for Timer 0 to generate square wave
           ORG   000BH        ;Timer 0 interrupt vector table
           CPL   P2.1         ;toggle P2.1 pin
           RETI               ;return from ISR
;
;--The main program for initialization
           ORG   0030H        ;after vector table space
MAIN:      MOV   TMOD,#02H    ;Timer 0, mode 2(auto-reload)
           MOV   P0,#0FFH     ;make P0 an input port
           MOV   TH0,#-92     ;TH0=A4H for -92
           MOV   IE,#82H      ;IE=10000010(bin) enable Timer 0
           SETB  TR0          ;Start Timer 0
BACK:      MOV   A,P0         ;get data from P0
           MOV   P1,A         ;issue it to P1
           SJMP  BACK         ;keep doing it
                              ;loop unless interrupted by TF0
           END
```

In Example 11-2, the interrupt service routine was short enough that it could be placed in memory locations allocated to the Timer 0 interrupt. However, that is not always the case. See Example 11-3.

Example 11-3

Rewrite Example 11-2 to create a square wave that has a high portion of 1085 μs and a low portion of 15 μs. Assume XTAL = 11.0592 MHz. Use Timer 1.

Solution:

Since 1085 μs is 1000 × 1.085 we need to use mode 1 of Timer 1.

```
;--Upon wake-up go to main, avoid using memory space
;--allocated to Interrupt Vector Table
           ORG   0000H
           LJMP  MAIN       ;bypass interrupt vector table
;
;--ISR for Timer 1 to generate square wave
           ORG   001BH      ;Timer 1 interrupt vector table
           LJMP  ISR_T1     ;jump to ISR
;
;--The main program for initialization
           ORG   0030H      ;after vector table
MAIN:      MOV   TMOD,#10H  ;Timer 1, mode 1
           MOV   P0,#0FFH   ;make P0 an input port
           MOV   TL1,#018H  ;TL1=18 the Low byte of -1000
           MOV   TH1,#0FCH  ;TH1=FC the High byte of -1000
           MOV   IE,#88H    ;IE=10001000 enable Timer 1 int.
           SETB  TR1        ;start Timer 1
BACK:      MOV   A,P0       ;get data from P0
           MOV   P1,A       ;issue it to P1
           SJMP  BACK       ;keep doing it
;
;--Timer 1 ISR. Must be reloaded since not auto-reload
ISR_T1:    CLR   TR1        ;stop Timer 1
           CLR   P2.1       ;P2.1=0, start of low portion
           MOV   R2,#4      ;                         2 MC
HERE:      DJNZ  R2,HERE    ;4x2 machine cycle(MC)    8 MC
           MOV   TL1,#18H   ;load T1 Low byte value   2 MC
           MOV   TH1,#0FCH  ;load T1 High byte value  2 MC
           SETB  TR1        ;starts Timer 1           1 MC
           SETB  P2.1       ;P2.1=1, back to high     1 MC
           RETI             ;return to main
           END
```

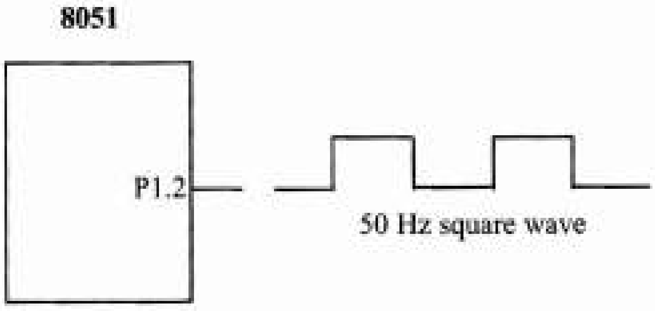
Notice that the low portion of the pulse is created by the 14 MC (machine cycles) where each MC = 1.085 μs and 14 × 1.085 μs = 15.19 μs.

Example 11-4

Write a program to generate a square wave of 50 Hz frequency on pin P1.2. This is similar to Example 9-12 except that it uses an interrupt for Timer 0. Assume that XTAL = 11.0592 MHz.

Solution:

```
           ORG   0
           LJMP  MAIN
           ORG   000BH            ;ISR for Timer 0
           CPL   P1.2             ;complement P1.2
           MOV   TL0,#00          ;reload timer values
           MOV   TH0,#0DCH
           RETI                   ;return from interrupt
           ORG   30H              ;starting location for prog.
;------main program for initialization
MAIN:      MOV   TMOD,#00000001B ;Timer 0, Mode 1
           MOV   TL0,#00
           MOV   TH0,#0DCH
           MOV   IE,#82H          ;enable Timer 0 interrupt
           SETB  TR0              ;start timer
HERE:      SJMP  HERE             ;stay here until interrupted
           END
```



Review Questions

1. True or false. There is only a single interrupt in the interrupt vector table assigned to both Timer 0 and Timer 1.
2. What address in the interrupt vector table is assigned to Timer 0?
3. Which bit of IE belongs to the timer interrupt? Show how both are enabled.
4. Assume that Timer 1 is programmed in mode 2, TH1 = F5H, and the IE bit for Timer 1 is enabled. Explain how the interrupt for the timer works.
5. True or false. The last two instructions of the ISR for Timer 0 are:

```
          CLR   TF0
          RETI
```

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

The 8051 has two external hardware interrupts. Pin 12 (P3.2) and pin 13 (P3.3) of the 8051, designated as INT0 and INT1, are used as external hardware interrupts. Upon activation of these pins, the 8051 gets interrupted in whatever it is doing and jumps to the vector table to perform the interrupt service routine. In this section we study these two external hardware interrupts of the 8051 with some examples.

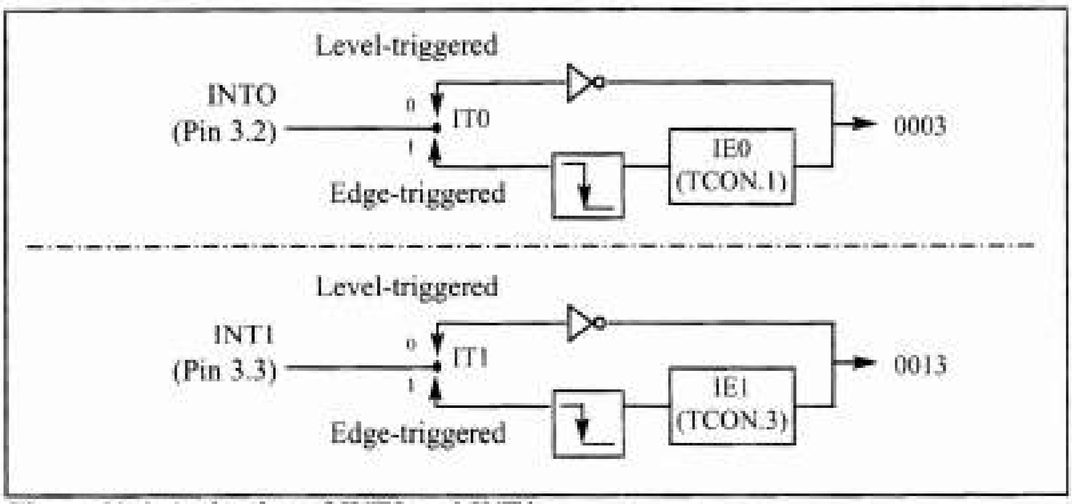


Figure 11-4. Activation of INT0 and INT1

External interrupts INT0 and INT1

There are only two external hardware interrupts in the 8051: INT0 and INT1. They are located on pins P3.2 and P3.3 of port 3, respectively. The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1, respectively. As mentioned in Section 11.1, they are enabled and disabled using the IE register. How are they activated? There are two types of activation for the external hardware interrupts: (1) level triggered, and (2) edge triggered. Let's look at each one. First, we see how the level-triggered interrupt works.

Level-triggered interrupt

In the level-triggered mode, INT0 and INT1 pins are normally high (just like all I/O port pins) and if a low-level signal is applied to them, it triggers the interrupt. Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt. This is called a *level-triggered* or *level-activated* interrupt and is the default mode upon reset of the 8051. The low-level signal at the INT pin must be removed before the execution of the last instruction of the interrupt service routine, RETI; otherwise, another interrupt will be generated. In other words, if the low-level interrupt signal is not removed before the ISR is finished it is interpreted as another interrupt and the 8051 jumps to the vector table to execute the ISR again. Look at Example 11-5.

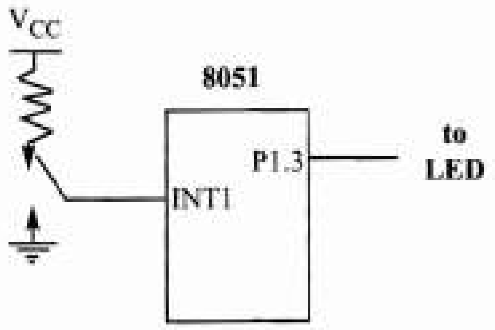
Example 11-5

Assume that the INT1 pin is connected to a switch that is normally high. Whenever it goes low, it should turn on an LED. The LED is connected to P1.3 and is normally off. When it is turned on it should stay on for a fraction of a second. As long as the switch is pressed low, the LED should stay on.

Solution:

```
           ORG   0000H
           LJMP  MAIN             ;bypass interrupt vector table
;--ISR for hardware interrupt INT1 to turn on the LED
           ORG   0013H            ;INT1 ISR
           SETB  P1.3             ;turn on LED
           MOV   R3,#255          ;load counter
BACK:      DJNZ  R3,BACK          ;keep LED on for a while
           CLR   P1.3             ;turn off the LED
           RETI                   ;return from ISR
;--MAIN program for initialization
           ORG   30H
MAIN:      MOV   IE,#10000100B    ;enable external INT1
HERE:      SJMP  HERE             ;stay here until interrupted
           END
```

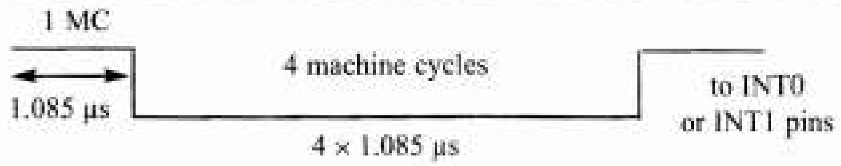
Pressing the switch will turn the LED on. If it is kept activated, the LED stays on.



In this program, the microcontroller is looping continuously in the HERE loop. Whenever the switch on INT1 (pin P3.3) is activated, the microcontroller gets out of the loop and jumps to vector location 0013H. The ISR for INT1 turns on the LED, keeps it on for a while, and turns it off before it returns. If by the time it executes the RETI instruction, the INT1 pin is still low, the microcontroller initiates the interrupt again. Therefore, to end this problem, the INT1 pin must be brought back to high by the time RETI is executed.

Sampling the low level-triggered interrupt

Pins P3.2 and P3.3 are used for normal I/O unless the INT0 and INT1 bits in the IE registers are enabled. After the hardware interrupts in the IE register are enabled, the controller keeps sampling the INT*n* pin for a low-level signal once each machine cycle. According to one manufacturer's data sheet "the pin must be held in a low state until the start of the execution of ISR. If the INT*n* pin is brought back to a logic high before the start of the execution of ISR there will be no interrupt." However, upon activation of the interrupt due to the low level, it must be brought back to high before the execution of RETI. Again, according to one manufacturer's data sheet, "If the INT*n* pin is left at a logic low after the RETI instruction of the ISR, another interrupt will be activated after one instruction is executed." Therefore, to ensure the activation of the hardware interrupt at the INT*n* pin, make sure that the duration of the low-level signal is around 4 machine cycles, but no more. This is due to the fact that the level-triggered interrupt is not latched. Thus the pin must be held in a low state until the start of the ISR execution.



Note: On RESET, IT0 (TCON.0) and IT1 (TCON.2) are both low, making external interrupts level-triggered.

Figure 11-5. Minimum Duration of the Low Level-Triggered Interrupt (XTAL = 11.0592 MHz)

Edge-triggered interrupts

As stated before, upon reset the 8051 makes INT0 and INT1 low-level triggered interrupts. To make them edge-triggered interrupts, we must program the bits of the TCON register. The TCON register holds, among other bits, the IT0 and IT1 flag bits that determine level- or edge-triggered mode of the hardware interrupts. IT0 and IT1 are bits D0 and D2 of the TCON register, respectively. They are also referred to as TCON.0 and TCON.2 since the TCON register is bit-addressable. Upon reset, TCON.0 (IT0) and TCON.2 (IT1) are both 0s, meaning that the external hardware interrupts of INT0 and INT1 pins are low-level triggered. By making the TCON.0 and TCON.2 bits high with instructions such as "`SETB  TCON.0`" and "`SETB  TCON.2`", the external hardware interrupts of INT0 and INT1 become edge-triggered. For example, the instruction "`SETB CON.2`" makes INT1 what is called an *edge-triggered interrupt*, in which, when a high-to-low signal is applied to pin P3.3, in this case, the controller will be interrupted and forced to jump to location 0013H in the vector table to service the ISR (assuming that the interrupt bit is enabled in the IE register).

D7							D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine.
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off.
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the service routine.
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off.
IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed. *Note:* This flag does not latch low-level triggered interrupts.
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt.
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when external interrupt (H-to-L transition) edge is detected. Cleared by CPU when interrupt is processed. *Note:* This flag does not latch low-level triggered interrupts.
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt.

Figure 11-6. TCON (Timer/Counter) Register (Bit-addressable)

Look at Example 11-6. Notice that the only difference between this program and the program in Example 11-5 is in the first line of MAIN where the instruction "`SETB TCON.2`" makes INT1 an edge-triggered interrupt. When the falling edge of the signal is applied to pin INT1, the LED will be turned on momentarily. The LED's on-state duration depends on the time delay inside the ISR for INT1. To turn on the LED again, another high-to-low pulse must be applied to pin 3.3. This is the opposite of Example 11-5. In Example 11-5, due to the level-triggered nature of the interrupt, as long as INT1 is kept at a low level, the LED is kept in the on state. But in this example, to turn on the LED again, the INT1 pulse must be brought back high and then forced low to create a falling edge to activate the interrupt.

Example 11-6

Assuming that pin 3.3 (INT1) is connected to a pulse generator, write a program in which the falling edge of the pulse will send a high to P1.3, which is connected to an LED (or buzzer). In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT1 pin. This is an edge-triggered version of Example 11-5.

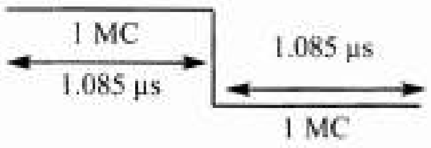
Solution:

```
      ORG   0000H
      LJMP  MAIN
;--ISR for hardware interrupt INT1 to turn on the LED
      ORG   0013H            ;INT1 ISR
      SETB  P1.3             ;turn on the LED
      MOV   R3,#255
BACK: DJNZ  R3,BACK          ;keep the LED on for a while
      CLR   P1.3             ;turn off the LED
      RETI                   ;return from ISR
;--MAIN program for initialization
      ORG   30H
MAIN: SETB  TCON.2           ;make INT1 edge-trigger interrupt
      MOV   IE,#10000100B    ;enable External INT1
HERE: SJMP  HERE             ;stay here until interrupted
      END
```

Sampling the edge-triggered interrupt

Before ending this section, we need to answer the question of how often the edge-triggered interrupt is sampled. In edge-triggered interrupts, the external source must be held high for at least one machine cycle, and then held low for at least one machine cycle to ensure that the transition is seen by the microcontroller.

Minimum pulse duration to detect edge-triggered interrupts.
XTAL = 11.0592 MHz



The falling edge is latched by the 8051 and is held by the TCON register. The TCON.1 and TCON.3 bits hold the latched falling edge of pins INT0 and INT1, respectively. TCON.1 and TCON.3 are also called IE0 and IE1, respectively, as shown in Figure 11-6. They function as interrupt-in-service flags. When an interrupt-in-service flag is raised, it indicates to the external world that the interrupt is being serviced and no new interrupt on this INT*n* pin will be responded to until this service is finished. This is just like the busy signal you get if calling a telephone number that is in use. Regarding the IT0 and IT1 bits in the TCON register, the following two points must be emphasized.

1. The first point is that when the ISRs are finished (that is, upon execution of instruction RETI), these bits (TCON.1 and TCON.3) are cleared, indicating that the interrupt is finished and the 8051 is ready to respond to another interrupt on that pin. For another interrupt to be recognized, the pin must go back to a logic high state and be brought back low to be considered an edge-triggered interrupt.
2. The second point is that while the interrupt service routine is being executed, the INT*n* pin is ignored, no matter how many times it makes a high-to-low transition. In reality one of the functions of the RETI instruction is to clear the corresponding bit in the TCON register (TCON.1 or TCON.3). This informs us that the service routine is no longer in progress and has finished being serviced. For this reason, TCON.1 and TCON.3 in the TCON register are called interrupt-in-service flags. The interrupt-in-service flag goes high whenever a falling edge is detected at the INT pin, and stays high during the entire execution of the ISR. It is only cleared by RETI, the last instruction of the ISR. Because of this, there is no need for an instruction such as "`CLR TCON.1`" (or "`CLR TCON.3`" for INT1) before the RETI in the ISR associated with the hardware interrupt INT0. As we will see in the next section, this is not the case for the serial interrupt.

Example 11-7

What is the difference between the RET and RETI instructions? Explain why we cannot use RET instead of RETI as the last instruction of an ISR.

Solution:

Both perform the same actions of popping off the top two bytes of the stack into the program counter, and making the 8051 return to where it left off. However, RETI also performs an additional task of clearing the interrupt-in-service flag, indicating that the servicing of the interrupt is over and the 8051 now can accept a new interrupt on that pin. If you use RET instead of RETI as the last instruction of the interrupt service routine, you simply block any new interrupt on that pin after the first interrupt, since the pin status would indicate that the interrupt is still being serviced. In the cases of TF0, TF1, TCON.1, and TCON.3, they are cleared by the execution of RETI.

More about the TCON register

Next we look at the TCON register more closely to understand its role in handling interrupts. Figure 11-6 shows the bits of the TCON register.

IT0 and IT1

TCON.0 and TCON.2 are referred to as IT0 and IT1, respectively. These two bits set the low-level or edge-triggered modes of the external hardware interrupts of the INT0 and INT1 pins. They are both 0 upon reset, which makes them low-level triggered. The programmer can make either of them high to make the external hardware interrupt edge-triggered. In a given system based on the 8051, once they are set to 0 or 1 they will not be altered again since the designer has fixed the interrupt as either edge- or level-triggered.

IE0 and IE1

TCON.1 and TCON.3 are referred to as IE0 and IE1, respectively. These bits are used by the 8051 to keep track of the edge-triggered interrupt only. In other words, if the IT0 and IT1 are 0, meaning that the hardware interrupts are low-level triggered, IE0 and IE1 are not used at all. The IE0 and IE1 bits are used by the 8051 only to latch the high-to-low edge transition on the INT0 and INT1 pins. Upon the edge transition pulse on the INT0 (or INT1) pin, the 8051 marks (sets high) the IE*x* bit in the TCON register, jumps to the vector in the interrupt vector table, and starts to execute the ISR. While it is executing the ISR, no H-to-L pulse transition on the INT0 (or INT1) is recognized, thereby preventing any interrupt inside the interrupt. Only the execution of the RETI instruction at the end of the ISR will clear the IE*x* bit, indicating that a new H-to-L pulse will activate the interrupt again. From this discussion we can see that the IE0 and IE1 bits are used internally by the 8051 to indicate whether or not an interrupt is in use. In other words, the programmer is not concerned with these bits since they are solely for internal use.

TR0 and TR1

These are the D4 (TCON.4) and D6 (TCON.6) bits of the TCON register. We were introduced to these bits in Chapter 9. They are used to start or stop timers 0 and 1, respectively. Although we have used syntax such as "`SETB TRx`" and "`CLR Trx`", we could have used instructions such as "`SETB TCON.4`" and "`CLR TCON.4`" since TCON is a bit-addressable register.

TF0 and TF1

These are the D5 (TCON.5) and D7 (TCON.7) bits of the TCON register. We were introduced to these bits in Chapter 9. They are used by timers 0 and 1, respectively, to indicate if the timer has rolled over. Although we have used the syntax "`JNB TFx,target`" and "`CLR Trx`", we could have used instructions such as "`JNB TCON.5,target`" and "`CLR TCON.5`" since TCON is bit-addressable.

Review Questions

1. True or false. There is a single interrupt in the interrupt vector table assigned to both external hardware interrupts IT0 and IT1.
2. What address in the interrupt vector table is assigned to INT0 and INT1? How about the pin numbers on port 3?
3. Which bit of IE belongs to the external hardware interrupts? Show how both are enabled.
4. Assume that the IE bit for the external hardware interrupt EX1 is enabled and is active low. Explain how this interrupt works when it is activated.
5. True or false. Upon reset, the external hardware interrupt is low-level triggered.
6. In Question 5, how do we make sure that a single interrupt is not recognized as multiple interrupts?
7. True or false. The last two instructions of the ISR for INT0 are:
```
        CLR   TCON.1
        RETI
```
8. Explain the role that each of the two bits TCON.0 and TCON.2 play in the execution of external interrupt 0.

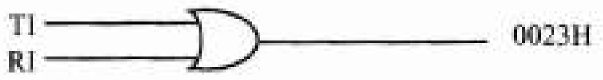
SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPT

In Chapter 10 we studied the serial communication of the 8051. All examples in that chapter used the polling method. In this section we explore interrupt-based serial communication, which allows the 8051 to do many things, in addition to sending and receiving data from the serial communication port.

RI and TI flags and interrupts

As you may recall from Chapter 10, TI (transfer interrupt) is raised when the last bit of the framed data, the stop bit, is transferred, indicating that the SBUF register is ready to transfer the next byte. RI (received interrupt), is raised when the entire frame of data, including the stop bit, is received. In other words, when the SBUF register has a byte, RI is raised to indicate that the received byte needs to be picked up before it is lost (overrun) by new incoming serial data. As far as serial communication is concerned, all the above concepts apply equally when using either polling or an interrupt. The only difference is in how the serial communication needs are served. In the polling method, we wait for the flag (TI or RI) to be raised; while we wait we cannot do anything else. In the interrupt method, we are notified when the 8051 has received a byte, or is ready to send the next byte; we can do other things while the serial communication needs are served.

In the 8051 only one interrupt is set aside for serial communication. This interrupt is used to both send and receive data. If the interrupt bit in the IE register (IE.4) is enabled, when RI or TI is raised the 8051 gets interrupted and jumps to memory address location 0023H to execute the ISR. In that ISR we must examine the TI and RI flags to see which one caused the interrupt and respond accordingly. See Example 11-8.



Serial interrupt is invoked by TI or RI flags

Figure 11-7. Single Interrupt for Both TI and RI

Use of serial COM in the 8051

In the vast majority of applications, the serial interrupt is used mainly for receiving data and is never used for sending data serially. This is like receiving a telephone call, where we need a ring to be notified. If we need to make a phone call there are other ways to remind ourselves and so no need for ringing. In receiving the phone call, however, we must respond immediately no matter what we are doing or we will miss the call. Similarly, we use the serial interrupt to receive incoming data so that it is not lost. Look at Example 11-9.

Example 11-8

Write a program in which the 8051 reads data from P1 and writes it to P2 continuously while giving a copy of it to the serial COM port to be transferred serially. Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

```
          ORG   0
          LJMP  MAIN
          ORG   23H
          LJMP  SERIAL          ;jump to serial interrupt ISR
          ORG   30H
MAIN:     MOV   P1,#0FFH        ;make P1 an input port
          MOV   TMOD,#20H       ;timer 1, mode 2(auto-reload)
          MOV   TH1,#0FDH       ;9600 baud rate
          MOV   SCON,#50H       ;8-bit, 1 stop, REN enabled
          MOV   IE,#10010000B   ;enable serial interrupt
          SETB  TR1             ;start timer 1
BACK:     MOV   A,P1            ;read data from port 1
          MOV   SBUF,A          ;give a copy to SBUF
          MOV   P2,A            ;send it to P2
          SJMP  BACK            ;stay in loop indefinitely
;
;------------------Serial Port ISR
          ORG   100H
SERIAL:   JB    TI,TRANS        ;jump if TI is high
          MOV   A,SBUF          ;otherwise due to receive
          CLR   RI              ;clear RI since CPU does not
          RETI                  ;return from ISR
TRANS:    CLR   TI              ;clear TI since CPU does not
          RETI                  ;return from ISR
          END
```

In the above program notice the role of TI and RI. The moment a byte is written into SBUF it is framed and transferred serially. As a result, when the last bit (stop bit) is transferred the TI is raised, which causes the serial interrupt to be invoked since the corresponding bit in the IE register is high. In the serial ISR, we check for both TI and RI since both could have invoked the interrupt. In other words, there is only one interrupt for both transmit and receive.

Clearing RI and TI before the RETI instruction

Notice in Example 11-9 that the last instruction before the RETI is the clearing of the RI or TI flags. This is necessary since there is only one interrupt for both receive and transmit, and the 8051 does not know who generated it; therefore, it is the job of the ISR to clear the flag. Contrast this with the external and timer interrupts where it is the job of the 8051 to clear the interrupt flags. By contrast,

Example 11-9

Write a program in which the 8051 gets data from P1 and sends it to P2 continuously while incoming data from the serial port is sent to P0. Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

Solution:

```
           ORG   0
           LJMP  MAIN
           ORG   23H
           LJMP  SERIAL           ;jump to serial ISR
           ORG   30H
MAIN:      MOV   P1,#0FFH         ;make P1 an input port
           MOV   TMOD,#20H        ;timer 1, mode 2(auto-reload)
           MOV   TH1,#0FDH        ;9600 baud rate
           MOV   SCON,#50H        ;8-bit,1 stop, REN enabled
           MOV   IE,#10010000B    ;enable serial interrupt
           SETB  TR1              ;start Timer 1
BACK:      MOV   A,P1             ;read data from port 1
           MOV   P2,A             ;send it to P2
           SJMP  BACK             ;stay in loop indefinitely
;------------------SERIAL PORT ISR
           ORG   100H
SERIAL:    JB    TI,TRANS         ;jump if TI is high
           MOV   A,SBUF           ;otherwise due to receive
           MOV   P0,A             ;send incoming data to P0
           CLR   RI               ;clear RI since CPU doesn't
           RETI                   ;return from ISR
TRANS:     CLR   TI               ;clear TI since CPU doesn't
           RETI                   ;return from ISR
           END
```

in serial communication the RI (or TI) must be cleared by the programmer using software instructions such as "`CLR TI`" and "`CLR RI`" in the ISR. See Example 11-10. Notice that the last two instructions of the ISR are clearing the flag, followed by RETI.

Before finishing this section notice the list of all interrupt flags given in Table 11-2. While the TCON register holds four of the interrupt flags, in the 8051 the SCON register has the RI and TI flags.

Table 11-2: Interrupt Flag Bits for the 8051/52

Interrupt	Flag	SFR Register Bit
External 0	IE0	TCON.1
External 1	IE1	TCON.3
Timer 0	TF0	TCON.5
Timer 1	TF1	TCON.7
Serial port	T1	SCON.1
Timer 2	TF2	T2CON.7 (AT89C52)
Timer 2	EXF2	T2CON.6 (AT89C52)

Example 11-10

Write a program using interrupts to do the following:
(a) Receive data serially and send it to P0,
(b) Have port P1 read and transmitted serially, and a copy given to P2,
(c) Make Timer 0 generate a square wave of 5 kHz frequency on P0.1.
Assume that XTAL = 11.0592 MHz. Set the baud rate at 4800.

Solution:

```
           ORG   0
           LJMP  MAIN
           ORG   000BH              ;ISR for Timer 0
           CPL   P0.1               ;toggle P0.1
           RETI                     ;return from ISR
           ORG   23H
           LJMP  SERIAL             ;jump to serial int. ISR
           ORG   30H
MAIN:      MOV   P1,#0FFH           ;make P1 an input port
           MOV   TMOD,#22H          ;timer 0&1,mode 2, auto-reload
           MOV   TH1,#0F6H          ;4800 baud rate
           MOV   SCON,#50H          ;8-bit, 1 stop, REN enabled
           MOV   TH0,#-92           ;for 5 KHz wave
           MOV   IE,#10010010B      ;enable serial, timer 0 int.
           SETB  TR1                ;start timer 1
           SETB  TR0                ;start timer 0
BACK:      MOV   A,P1               ;read data from port 1
           MOV   SBUF,A             ;give a copy to SBUF
           MOV   P2,A               ;write it to P2
           SJMP  BACK               ;stay in loop indefinitely

;-------------------SERIAL PORT ISR
           ORG   100H
SERIAL:    JB    TI,TRANS           ;jump if TI is high
           MOV   A,SBUF             ;otherwise due to received
           MOV   P0,A               ;send serial data to P0
           CLR   RI                 ;clear RI since CPU does not
           RETI                     ;return from ISR
TRANS:     CLR   TI                 ;clear TI since CPU does not
           RETI                     ;return from ISR
           END
```

Review Questions

1. True or false. There is a single interrupt in the interrupt vector table assigned to both the TI and RI interrupts.
2. What address in the interrupt vector table is assigned to the serial interrupt?
3. Which bit of the IE register belongs to the serial interrupt? Show how it is enabled.
4. Assume that the IE bit for the serial interrupt is enabled. Explain how this interrupt gets activated and also explain its actions upon activation.

5. True or false. Upon reset, the serial interrupt is active and ready to go.
6. True or false. The last two instructions of the ISR for the receive interrupt are:
```
        CLR   RI
        RETI
```
7. Answer Question 6 for the send interrupt.

SECTION 11.5: INTERRUPT PRIORITY IN THE 8051/52

The next topic that we must deal with is what happens if two interrupts are activated at the same time? Which of these two interrupts is responded to first? Interrupt priority is the main topic of discussion in this section.

Interrupt priority upon reset

When the 8051 is powered up, the priorities are assigned according to Table 11-3. From Table 11-3 we see, for example, that if external hardware interrupts 0 and 1 are activated at the same time, external interrupt 0 (INT0) is responded to first. Only after INT0 has been serviced is INT1 serviced, since INT1 has the lower priority. In reality, the priority scheme in the table is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed in Table 11-3, and responds accordingly.

Table 11-3: 8051/52 Interrupt Priority Upon Reset

Highest to Lowest Priority	
External Interrupt 0	(INT0)
Timer Interrupt 0	(TF0)
External Interrupt 1	(INT1)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)
Timer 2 (8052 only)	**TF2**

Example 11-11

Discuss what happens if interrupts INT0, TF0, and INT1 are activated at the same time. Assume priority levels were set by the power-up reset and that the external hardware interrupts are edge-triggered.

Solution:

If these three interrupts are activated at the same time, they are latched and kept internally. Then the 8051 checks all five interrupts according to the sequence listed in Table 11-3. If any is activated, it services it in sequence. Therefore, when the above three interrupts are activated, IE0 (external interrupt 0) is serviced first, then Timer 0 (TF0), and finally IE1 (external interrupt 1).

D7							D0
--	--	PT2	PS	PT1	PX1	PT0	PX0

Priority bit = 1 assigns high priority. Priority bit = 0 assigns low priority.

--	IP.7	Reserved
--	IP.6	Reserved
PT2	IP.5	Timer 2 interrupt priority bit (8052 only)
PS	IP.4	Serial port interrupt priority bit
PT1	IP.3	Timer 1 interrupt priority bit
PX1	IP.2	External interrupt 1 priority bit
PT0	IP.1	Timer 0 interrupt priority bit
PX0	IP.0	External interrupt 0 priority bit

User software should never write 1s to unimplemented bits, since they may be used in future products.

Figure 11-8. Interrupt Priority Register (Bit-addressable)

Setting interrupt priority with the IP register

We can alter the sequence of Table 11-3 by assigning a higher priority to any one of the interrupts. This is done by programming a register called IP (interrupt priority). Figure 11-8 shows the bits of the IP register. Upon power-up reset, the IP register contains all 0s, making the priority sequence based on Table 11-3. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high. Look at Example 11-12.

Example 11-12

(a) Program the IP register to assign the highest priority to INT1 (external interrupt 1), then (b) discuss what happens if INT0, INT1, and TF0 are activated at the same time. Assume that the interrupts are both edge-triggered.

Solution:

(a)
```
MOV IP,#00000100B ;IP.2=1 to assign INT1 higher priority
```
The instruction "`SETB IP.2`" also will do the same thing as the above line since IP is bit-addressable.

(b) The instruction in Step (a) assigned a higher priority to INT1 than the others; therefore, when INT0, INT1, and TF0 interrupts are activated at the same time, the 8051 services INT1 first, then it services INT0, then TF0. This is due to the fact that INT1 has a higher priority than the other two because of the instruction in Step (a). The instruction in Step (a) makes both the INT0 and TF0 bits in the IP register 0. As a result, the sequence in Table 11-3 is followed, which gives a higher priority to INT0 over TF0.

Example 11-13

Assume that after reset, the interrupt priority is set by the instruction "`MOV IP, #00001100B`". Discuss the sequence in which the interrupts are serviced.

Solution:

The instruction "`MOV IP,#00001100B`" (B is for binary) sets the external interrupt 1 (INT1) and Timer 1 (TF1) to a higher priority level compared with the rest of the interrupts. However, since they are polled according to Table 11-3, they will have the following priority.

Highest Priority	External Interrupt 1	(INT1)
	Timer Interrupt 1	(TF1)
	External Interrupt 0	(INT0)
	Timer Interrupt 0	(TF0)
Lowest Priority	Serial Communication	(RI + TI)

Another point that needs to be clarified is the interrupt priority when two or more interrupt bits in the IP register are set to high. In this case, while these interrupts have a higher priority than others, they are serviced according to the sequence of Table 11-3. See Example 11-13.

Interrupt inside an interrupt

What happens if the 8051 is executing an ISR belonging to an interrupt and another interrupt is activated? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is an interrupt inside an interrupt. In the 8051 a low-priority interrupt can be interrupted by a higher-priority interrupt, but not by another low-priority interrupt. Although all the interrupts are latched and kept internally, no low-priority interrupt can get the immediate attention of the CPU until the 8051 has finished servicing the high-priority interrupts.

Triggering the interrupt by software

There are times when we need to test an ISR by way of simulation. This can be done with simple instructions to set the interrupts high and thereby cause the 8051 to jump to the interrupt vector table. For example, if the IE bit for Timer 1 is set, an instruction such as "`SETB TF1`" will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table. In other words, we do not need to wait for Timer 1 to roll over to have an interrupt. We can cause an interrupt with an instruction that raises the interrupt flag.

Review Questions

1. True or false. Upon reset, all interrupts have the same priority.
2. What register keeps track of interrupt priority in the 8051? Is it a bit-addressable register?
3. Which bit of IP belongs to the serial interrupt priority? Show how to assign it the highest priority.
4. Assume that the IP register contains all 0s. Explain what happens if both INT0 and INT1 are activated at the same time.
5. Explain what happens if a higher-priority interrupt is activated while the 8051 is serving a lower-priority interrupt (that is, executing a lower-priority ISR).

SECTION 11.6: INTERRUPT PROGRAMMING IN C

So far all the programs in this chapter have been written in Assembly. In this section we show how to program the 8051/52's interrupts in 8051 C language. In reading this section, it is assumed that you already know the material in the first two sections of this chapter.

8051 C interrupt numbers

The 8051 C compilers have extensive support for the 8051 interrupts with two major features as follows:

1. They assign a unique number to each of the 8051 interrupts, as shown in Table 11-4.
2. It can also assign a register bank to an ISR. This avoids code overhead due to the pushes and pops of the R0 - R7 registers.

Table 11-4: 8051/52 Interrupt Numbers in C

Interrupt	Name	Numbers used by 8051 C
External Interrupt 0	(INT0)	0
Timer Interrupt 0	(TF0)	1
External Interrupt 1	(INT1)	2
Timer Interrupt 1	(TF1)	3
Serial Communication	(RI + TI)	4
Timer 2 (8052 only)	(TF2)	5

Example 11-14 shows how a simple interrupt is written in 8051 C.

Example 11-14

Write a C program that continuously gets a single bit of data from P1.7 and sends it to P1.0, while simultaneously creating a square wave of 200 μs period on pin P2.5. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

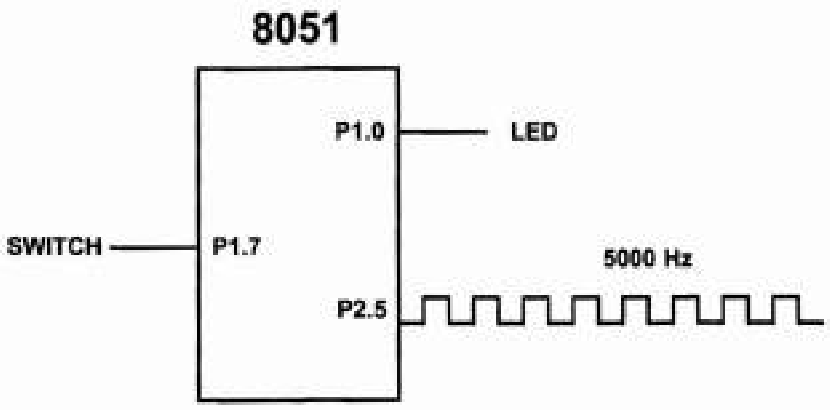
Solution:

We will use timer 0 in mode 2 (auto-reload). One half of the period is 100 μs. 100 /1. 085 μs = 92, and TH0 = 256 – 92 = 164 or A4H

```
#include <reg51.h>

sbit SW    = P1^7;
sbit IND   = P1^0;
sbit WAVE  = P2^5;

void timer0(void) interrupt 1
  {
    WAVE = ~WAVE;      //toggle pin
  }

void main()
  {
    SW = 1;            //make switch input
    TMOD = 0x02;
    TH0 = 0xA4;        //TH0 = -92
    IE = 0x82;         //enable interrupts for timer 0
    while(1)
      {
        IND = SW;      //send switch to LED
      }
  }
```

200 μs / 2 = 100 μs

100 μs / 1.085 μs = 92



Example 11-15

Write a C program that continuously gets a single bit of data from P1.7 and sends it to P1.0 in the main, while simultaneously (a) creating a square wave of 200 µs period on pin P2.5, and (b) sending letter 'A' to the serial port. Use Timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz. Use the 9600 baud rate.

Solution:

We will use Timer 0 in mode 2 (auto-reload). TH0 = 100/1.085 µs = –92, which is A4H

```
#include <reg51.h>

sbit SW     = P1^7;
sbit IND    = P1^0;
sbit WAVE   = P2^5;

void timer0(void) interrupt 1
  {
    WAVE = ~WAVE;      //toggle pin
  }

void serial0() interrupt 4
  {
    if(TI == 1)
      {
        SBUF = 'A';    //send A to serial port
        TI = 0;        //clear interrupt
      }
    else
      {
        RI = 0;        //clear interrupt
      }
  }

void main()
  {
    SW = 1;            //make switch input
    TH1 = -3;          //9600 baud
    TMOD = 0x22;       //mode 2 for both timers
    TH0 = 0xA4;        //-92=A4H for timer 0
    SCON = 0x50;
    TR0 = 1;
    TR1 = 1;                //start timer
    IE = 0x92;         //enable interrupt for T0
    while(1)           //stay here
      {
        IND = SW;      //send switch to LED
      }
  }
```

Example 11-16

Write a C program using interrupts to do the following:
(a) Receive data serially and send it to P0,
(b) Read port P1, transmit data serially, and give a copy to P2,
(c) Make timer 0 generate a square wave of 5 kHz frequency on P0.1.
Assume that XTAL = 11.0592 MHz. Set the baud rate at 4800.

Solution:

```
#include <reg51.h>
sbit WAVE = P0^1;

void timer0() interrupt 1
  {
    WAVE = ~WAVE;             //toggle pin
  }

void serial0() interrupt 4
  {
    if(TI == 1)
      {
        TI = 0;               //clear interrupt
      }
    else
      {
        P0 = SBUF;            //put value on pins
        RI = 0;               //clear interrupt
      }
  }

void main()
  {
    unsigned char x;
    P1 = 0xFF;                //make P1 an input
    TMOD = 0x22;
    TH1 = 0xF6;               //4800 baud rate
    SCON = 0x50;
    TH0 = 0xA4;               //5 kHz has T = 200 µs
    IE = 0x92;                //enable interrupts
    TR1 = 1;                  //start timer 1
    TR0 = 1;                  //start timer 0
    while(1)
      {
        x = P1;               //read value from pins
        SBUF = x;             //put value in buffer
        P2 = x;               //write value to pins
      }
  }
```

Example 11-17

Write a C program using interrupts to do the following:
(a) Generate a 10000 Hz frequency on P2.1 using T0 8-bit auto-reload,
(b) Use timer 1 as an event counter to count up a 1-Hz pulse and display it on P0. The pulse is connected to EX1.

Assume that XTAL = 11.0592 MHz. Set the baud rate at 9600.

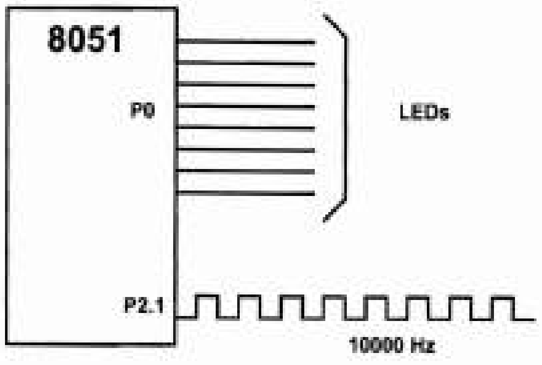
Solution:

```
#include <reg51.h>

sbit WAVE = P2^1;
unsigned char cnt;

void timer0() interrupt 1
  {
    WAVE = ~WAVE;             //toggle pin
  }
void timer1() interrupt 3
  {
    cnt++;                    //increment counter
    P0 = cnt;                 //display value on pins
  }

void main()
  {
    cnt = 0;                  //set counter to zero
    TMOD = 0x42;
    TH0 = 0x-46;              //10000 Hz
    IE = 0x86;                //enable interrupts
    TR0 = 1;                  //start timer 0
    TR1 = 1;                  //start timer 1
    while(1);                 //wait until interrupted
  }
```

1 / 10000 Hz = 100 μs

100 μs / 2 = 50 μs

50 μs / 1.085 μs = 46



SUMMARY

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service. Every interrupt has a program associated with it called the ISR, or interrupt service routine. The 8051 has 6 interrupts, 5 of which are user-accessible. The interrupts are for reset: two for the timers, two for external hardware interrupts, and a serial communication interrupt. The 8052 has an additional interrupt for Timer 2.

The 8051 can be programmed to enable or disable an interrupt, and the interrupt priority can be altered. This chapter showed how to program 8051/52 interrupts in both Assembly and C languages.

PROBLEMS

SECTION 11.1: 8051 INTERRUPTS

1. Which technique, interrupt or polling, avoids tying down the microcontroller?
2. Including reset, how many interrupts does the 8051 have?
3. In the 8051 what memory area is assigned to the interrupt vector table?
4. True or false. The 8051 programmer cannot change the memory space assigned to the interrupt vector table.
5. What memory address in the interrupt vector table is assigned to INT0?
6. What memory address in the interrupt vector table is assigned to INT1?
7. What memory address in the interrupt vector table is assigned to Timer 0?
8. What memory address in the interrupt vector table is assigned to Timer 1?
9. What memory address in the interrupt vector table is assigned to the serial COM interrupt?
10. Why do we put an LJMP instruction at address 0?
11. What are the contents of the IE register upon reset, and what do these values mean?
12. Show the instruction to enable the EX1 and Timer 1 interrupts.
13. Show the instruction to enable every interrupt of the 8051.
14. Which pin of the 8051 is assigned to the external hardware interrupts INT0 and INT1?
15. How many bytes of address space in the interrupt vector table are assigned to the INT0 and INT1 interrupts?
16. How many bytes of address space in the interrupt vector table are assigned to the Timer 0 and Timer 1 interrupts?
17. To put the entire interrupt service routine in the interrupt vector table, it must be no more than _____ bytes in size.
18. True or false. The IE register is not a bit-addressable register.
19. With a single instruction, show how to disable all the interrupts.
20. With a single instruction, show how to disable the EX1 interrupt.
21. True or false. Upon reset, all interrupts are enabled by the 8051.
22. In the 8051, how many bytes of ROM space are assigned to the reset interrupt, and why?

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

23. True or false. For both Timer 0 and Timer 1, there is an interrupt assigned to it in the interrupt vector table.
24. What address in the interrupt vector table is assigned to Timer 1?
25. Which bit of IE belongs to the Timer 0 interrupt? Show how it is enabled.
26. Which bit of IE belongs to the Timer 1 interrupt? Show how it is enabled.
27. Assume that Timer 0 is programmed in mode 2, TH1 = F0H, and the IE bit for Timer 0 is enabled. Explain how the interrupt for the timer works.
28. True or false. The last two instructions of the ISR for Timer 1 are:

```
            CLR   TF1
            RETI
```

29. Assume that Timer 1 is programmed for mode 1, TH0 = FFH, TL1 = F8H, and the IE bit for Timer 1 is enabled. Explain how the interrupt is activated.
30. If Timer 1 is programmed for interrupts in mode 2, explain when the interrupt is activated.
31. Write a program to create a square wave of T = 160 ms on pin P2.2 while at the same time the 8051 is sending out 55H and AAH to P1 continuously.
32. Write a program in which every 2 seconds, the LED connected to P2.7 is turned on and off four times, while at the same time the 8051 is getting data from P1 and sending it to P0 continuously. Make sure the on and off states are 50 ms in duration.

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

33. True or false. A single interrupt is assigned to each of the external hardware interrupts EX0 and EX1.
34. What address in the interrupt vector table is assigned to INT0 and INT1? How about the pin numbers on port 3?
35. Which bit of IE belongs to the EX0 interrupt? Show how it is enabled.
36. Which bit of IE belongs to the EX1 interrupt? Show how it is enabled.
37. Show how to enable both external hardware interrupts.
38. Assume that the IE bit for external hardware interrupt EX0 is enabled and is low-level triggered. Explain how this interrupt works when it is activated. How can we make sure that a single interrupt is not interpreted as multiple interrupts?
39. True or false. Upon reset, the external hardware interrupt is edge-triggered.
40. In Question 39, how do we make sure that a single interrupt is not recognized as multiple interrupts?
41. Which bits of TCON belong to EX0?
42. Which bits of TCON belong to EX1?
43. True or false. The last two instructions of the ISR for INT1 are:

```
    CLR   TCON.3
    RETI
```

44. Explain the role of TCON.0 and TCON.2 in the execution of external interrupt 0.
45. Explain the role of TCON.1 and TCON.3 in the execution of external interrupt 1.
46. Assume that the IE bit for external hardware interrupt EX1 is enabled and is edge-triggered. Explain how this interrupt works when it is activated. How can

we make sure that a single interrupt is not interpreted as multiple interrupts?

47. Write a program using interrupts to get data from P1 and send it to P2 while Timer 0 is generating a square wave of 3 kHz.
48. Write a program using interrupts to get data from P1 and send it to P2 while Timer 1 is turning on and off the LED connected to P0.4 every second.
49. Explain the difference between the low-level and edge-triggered interrupts.
50. How do we make the hardware interrupt edge-triggered?
51. Which interrupts are latched, low-level or edge-triggered?
52. Which register keeps the latched interrupt for INT0 and INT1?

SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPT

53. True or false. There are two interrupts assigned to interrupts TI and RI.
54. What address in the interrupt vector table is assigned to the serial interrupt? How many bytes are assigned to it?
55. Which bit of the IE register belongs to the serial interrupt? Show how it is enabled.
56. Assume that the IE bit for the serial interrupt is enabled. Explain how this interrupt gets activated and also explain its working upon activation.
57. True or false. Upon reset, the serial interrupt is blocked.
58. True or false. The last two instructions of the ISR for the receive interrupt are:
```
        CLR   TI
        RETI
```
59. Answer Question 58 for the receive interrupt.
60. Assuming that the interrupt bit in the IE register is enabled, when TI is raised, what happens subsequently?
61. Assuming that the interrupt bit in the IE register is enabled, when RI is raised, what happens subsequently?
62. Write a program using interrupts to get data serially and send it to P2 while at the same time Timer 0 is generating a square wave of 5 kHz.
63. Write a program using interrupts to get data serially and send it to P2 while Timer 0 is turning the LED connected to P1.6 on and off every second.

SECTION 11.5: INTERRUPT PRIORITY IN THE 8051/52

64. True or false. Upon reset, EX1 has the highest priority.
65. What register keeps track of interrupt priority in the 8051? Explain its role.
66. Which bit of IP belongs to the EX2 interrupt priority? Show how to assign it the highest priority.
67. Which bit of IP belongs to the Timer 1 interrupt priority? Show how to assign it the highest priority.
68. Which bit of IP belongs to the EX1 interrupt priority? Show how to assign it the highest priority.
69. Assume that the IP register has all 0s. Explain what happens if both INT0 and INT1 are activated at the same time.
70. Assume that the IP register has all 0s. Explain what happens if both TF0 and TF1 are activated at the same time.

71. If both TF0 and TF1 in the IP are set to high, what happens if both are activated at the same time?
72. If both INT0 and INT1 in the IP are set to high, what happens if both are activated at the same time?
73. Explain what happens if a low-priority interrupt is activated while the 8051 is serving a higher-priority interrupt.

ANSWERS TO REVIEW QUESTIONS

SECTION 11.1: 8051 INTERRUPTS

1. Interrupts
2. 5
3. Address locations 0000 to 25H. No. They are set when the processor is designed.
4. All 0s means that all interrupts are masked, and as a result no interrupts will be responded to by the 8051.
5. `MOV IE,#10000011B`
6. P3.3, which is pin 13 on the 40-pin DIP package
7. 0013H for INT1 and 001BH for Timer 1

SECTION 11.2: PROGRAMMING TIMER INTERRUPTS

1. False. There is an interrupt for each of the timers, Timer 0 and Timer 1.
2. 000BH
3. Bits D1 and D3 and "`MOV IE,#10001010B`" will enable both of the timer interrupts.
4. After Timer 1 is started with instruction "`SETB TR1`", the timer will count up from F5H to FFH on its own while the 8051 is executing other tasks. Upon rolling over from FFH to 00, the TF1 flag is raised, which will interrupt the 8051 in whatever it is doing and force it to jump to memory location 001BH to execute the ISR belonging to this interrupt.
5. False. There is no need for "`CLR TF0`" since the RETI instruction does that for us.

SECTION 11.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS

1. False. There is an interrupt for each of the external hardware interrupts of INT0 and INT1.
2. 0003H and 0013H. The pins numbered 12 (P3.2) and 13 (P3.3) on the DIP package.
3. Bits D0 and D2 and "`MOV IE,#10000101B`" will enable both of the external hardware interrupts.
4. Upon application of a low pulse (4 machine cycles wide) to pin P3.3, the 8051 is interrupted in whatever it is doing and jumps to ROM location 0013H to execute the ISR.
5. True
6. Make sure that the low pulse applied to pin INT1 is no wider than 4 machine cycles. Or, make sure that the INT1 pin is brought back to high by the time the 8051 executes the RETI instruction in the ISR.
7. False. There is no need for the "`CLR TCON.0`" since the RETI instruction does that for us.
8. TCON.0 is set to high to make INT0 an edge-triggered interrupt. If INT0 is edge-triggered (that is, TCON.0 is set), whenever a high-to-low pulse is applied to the INT0 pin it is captured (latched) and kept by the TCON.2 bit by making TCON.2 high. While the ISR for INT0 is being serviced, TCON.2 stays high no matter how many times an H-to-L pulse is applied to pin INT0. Upon the execution of the last instruction of the ISR, which is RETI, the TCON.2 bit is cleared, indicating that the INT0 pin can respond to another interrupt.

SECTION 11.4: PROGRAMMING THE SERIAL COMMUNICATION INTERRUPT

1. True. There is only one interrupt for both the transfer and receive.
2. 23H
3. Bit D4 (IE.4) and "`MOV IE,#10010000B`" will enable the serial interrupt.
4. The RI (received interrupt) flag is raised when the entire frame of data, including the stop bit, is received. As a result the received byte is delivered to the SBUF register and the 8051 jumps to memory location 0023H to execute the ISR belonging to this interrupt. In the serial COM interrupt service routine, we must save the SBUF contents before it is lost by the incoming data.
5. False
6. True. We must do it since the RETI instruction will not do it for the serial interrupt.
7.
```
    CLR   TI
    RETI
```

SECTION 11.5: INTERRUPT PRIORITY IN THE 8051/52

1. False. They are assigned priority according to Table 11-3.
2. IP (interrupt priority) register. Yes, it is bit-addressable.
3. Bit D4 (IP.4) and the instruction "`MOV IP,#00010000B`" will do it.
4. If both are activated at the same time, INT0 is serviced first since it has a higher priority. After INT0 is serviced, INT1 is serviced, assuming that the external interrupts are edge-triggered and H-to-L transitions are latched. In the case of low-level triggered interrupts, if both are activated at the same time, the INT0 is serviced first; then after the 8051 has finished servicing the INT0, it scans the INT0 and INT1 pins again, and if the INT1 pin is still high, it will be serviced.
5. We have an interrupt inside an interrupt, meaning that the lower-priority interrupt is put on hold and the higher one is serviced. After servicing this higher-priority interrupt, the 8051 resumes servicing the lower-priority ISR.

CHAPTER 12

LCD AND KEYBOARD INTERFACING

OBJECTIVES

Upon completion of this chapter, you will be able to:

**>>	List reasons that LCDs are gaining widespread use, replacing LEDs**
**>>	Describe the functions of the pins of a typical LCD**
**>>	List instruction command codes for programming an LCD**
**>>	Interface an LCD to the 8051**
**>>	Program an LCD in Assembly and C**
**>>	Explain the basic operation of a keyboard**
**>>	Describe the key press and detection mechanisms**
**>>	Interface a 4x4 keypad to the 8051 using C and Assembly**

This chapter explores some real-world applications of the 8051. We explain how to interface the 8051 to devices such as an LCD and a keyboard. In Section 12.1, we show LCD interfacing with the 8051. In Section 12.2, keyboard interfacing with the 8051 is shown. We use C and Assembly for both sections.

SECTION 12.1: LCD INTERFACING

This section describes the operation modes of LCDs, then describes how to program and interface an LCD to an 8051 using Assembly and C.

LCD operation

In recent years the LCD is finding widespread use replacing LEDs (seven-segment LEDs or other multisegment LEDs). This is due to the following reasons:

1. The declining prices of LCDs.
2. The ability to display numbers, characters, and graphics. This is in contrast to LEDs, which are limited to numbers and a few characters.
3. Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD. In contrast, the LED must be refreshed by the CPU (or in some other way) to keep displaying the data.
4. Ease of programming for characters and graphics.

LCD pin descriptions

The LCD discussed in this section has 14 pins. The function of each pin is given in Table 12-1. Figure 12-1 shows the pin positions for various LCDs.

V_{CC}, V_{SS}, and V_{EE}

While V_{CC} and V_{SS} provide +5V and ground, respectively, V_{EE} is used for controlling LCD contrast.

RS, register select

There are two very important registers inside the LCD. The RS pin is used for their selection as follows. If RS = 0, the instruction command code register is selected, allowing the user to send a command such as clear display, cursor at home, etc. If RS = 1 the data register is selected, allowing the user to send data to be displayed on the LCD.

R/W, read/write

R/W input allows the user to write information to the LCD or read information from it. R/W = 1 when reading; R/W = 0 when writing.

E, enable

The enable pin is used by the LCD to latch information presented to its data pins.

Table 12-1: Pin Descriptions for LCD

Pin	Symbol	I/O	Description
1	V_{SS}	--	Ground
2	V_{CC}	--	+5V power supply
3	V_{EE}	--	Power supply to control contrast
4	RS	I	RS = 0 to select command register, RS = 1 to select data register
5	R/W	I	R/W = 0 for write, R/W = 1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

When data is supplied to data pins, a high-to-low pulse must be applied to this pin in order for the LCD to latch in the data present at the data pins. This pulse must be a minimum of 450 ns wide.

D0 - D7

The 8-bit data pins, D0 - D7, are used to send information to the LCD or read the contents of the LCD's internal registers.

To display letters and numbers, we send ASCII codes for the letters A - Z, a - z, and numbers 0 - 9 to these pins while making RS = 1.

There are also instruction command codes that can be sent to the LCD to clear the display or force the cursor to the home position or blink the cursor. Table 12-2 lists the instruction command codes.

We also use RS = 0 to check the busy flag bit to see if the LCD is ready to receive information. The busy flag is D7 and can be read when R/W = 1 and RS = 0, as follows: if R/W = 1, RS = 0. When D7 = 1 (busy flag = 1), the LCD is busy taking care of internal operations and will not accept any new information. When D7 = 0, the LCD is ready to receive new information. *Note:* It is recommended to check the busy flag before writing any data to the LCD.

Table 12-2: LCD Command Codes

Code (Hex)	**Command to LCD Instruction Register**
1	Clear display screen
2	Return home
4	Decrement cursor (shift cursor to left)
6	Increment cursor (shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, cursor off
A	Display off, cursor on
C	Display on, cursor off
E	Display on, cursor blinking
F	Display on, cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift the entire display to the left
1C	Shift the entire display to the right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5x7 matrix

Note: This table is extracted from Table 12-4.

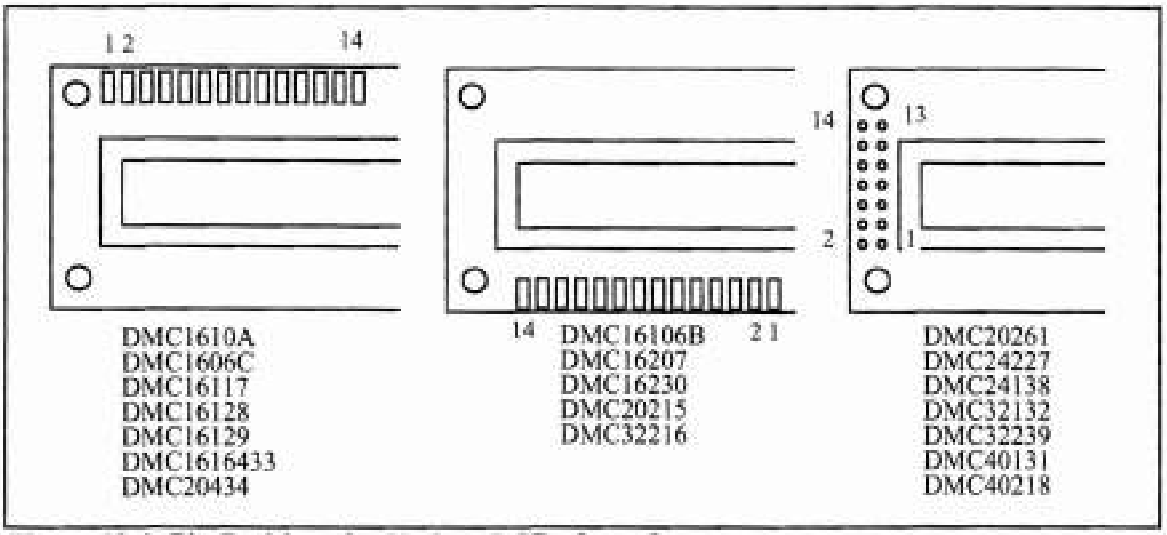


Figure 12-1. Pin Positions for Various LCDs from Optrex

Sending commands and data to LCDs with a time delay

To send any of the commands from Table 12-2 to the LCD, make pin RS = 0. For data, make RS = 1. Then send a high-to-low pulse to the E pin to enable the internal latch of the LCD. This is shown in Program 12-1. See Figure 12-2 for LCD connections.

```
;calls a time delay before sending next data/command
; P1.0-P1.7 are connected to LCD data pins D0-D7
; P2.0 is connected to RS pin of LCD
; P2.1 is connected to R/W pin of LCD
; P2.2 is connected to E pin of LCD
        ORG    0H
        MOV    A,#38H          ;init. LCD 2 lines,5x7 matrix
        ACALL  COMNWRT         ;call command subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#0EH          ;display on, cursor on
        ACALL  COMNWRT         ;call command subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#01           ;clear LCD
        ACALL  COMNWRT         ;call command subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#06H          ;shift cursor right
        ACALL  COMNWRT         ;call command subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#84H          ;cursor at line 1,pos. 4
        ACALL  COMNWRT         ;call command subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#'N'          ;display letter N
        ACALL  DATAWRT         ;call display subroutine
        ACALL  DELAY           ;give LCD some time
        MOV    A,#'O'          ;display letter O
        ACALL  DATAWRT         ;call display subroutine
AGAIN:  SJMP   AGAIN           ;stay here
COMNWRT:                       ;send command to LCD
        MOV    P1,A            ;copy reg A to port1
        CLR    P2.0            ;RS=0 for command
        CLR    P2.1            ;R/W=0 for write
        SETB   P2.2            ;E=1 for high pulse
        ACALL  DELAY           ;give LCD some time
        CLR    P2.2            ;E=0 for H-to-L pulse
        RET
DATAWRT:                       ;write data to LCD
        MOV    P1,A            ;copy reg A to port1
        SETB   P2.0            ;RS=1 for data
        CLR    P2.1            ;R/W=0 for write
        SETB   P2.2            ;E=1 for high pulse
        ACALL  DELAY           ;give LCD some time
        CLR    P2.2            ;E=0 for H-to-L pulse
        RET
```

Program 12-1: Communicating with LCD using a delay *(continued on next page)*

```
DELAY:    MOV   R3,#50          ;50 or higher for fast CPUs
HERE2:    MOV   R4,#255         ;R4=255
HERE:     DJNZ  R4,HERE         ;stay until R4 becomes 0
          DJNZ  R3,HERE2
          RET
          END
```

Program 12-1. *(continued from previous page)*

Sending code or data to the LCD with checking busy flag

The above code showed how to send commands to the LCD without checking the busy flag. Notice that we must put a long delay between issuing data or commands to the LCD. However, a much better way is to monitor the busy flag before issuing a command or data to the LCD. This is shown in Program 12-2.

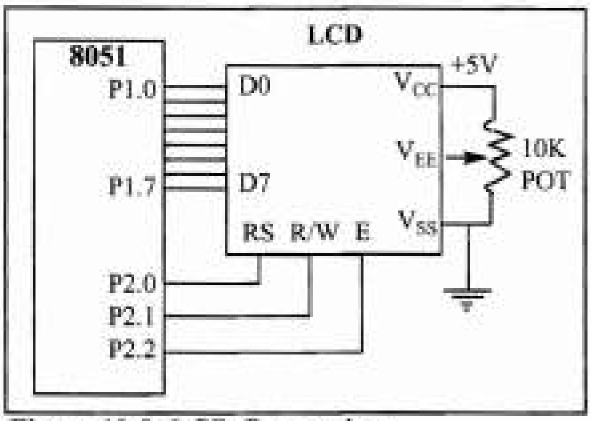


Figure 12-2. LCD Connections

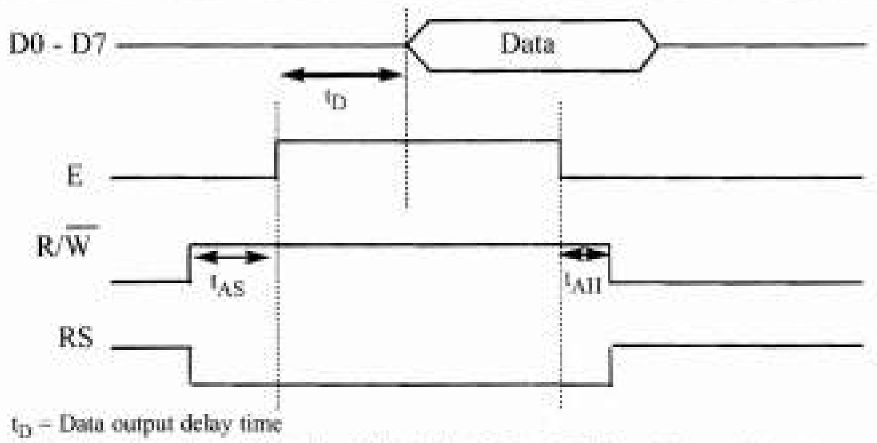
```
;Check busy flag before sending data, command to LCD
;P1=data pin,P2.0=RS,P2.1=R/W,P2.2=E pins
          MOV   A,#38H          ;init. LCD 2 lines,5x7 matrix
          ACALL COMMAND         ;issue command
          MOV   A,#0EH          ;LCD on, cursor on
          ACALL COMMAND         ;issue command
          MOV   A,#01H          ;clear LCD command
          ACALL COMMAND         ;issue command
          MOV   A,#06H          ;shift cursor right
          ACALL COMMAND         ;issue command
          MOV   A,#86H          ;cursor: line 1, pos. 6
          ACALL COMMAND         ;command subroutine
          MOV   A,#'N'          ;display letter N
          ACALL DATA_DISPLAY
          MOV   A,#'O'          ;display letter O
          ACALL DATA_DISPLAY
HERE:     SJMP  HERE            ;STAY HERE
COMMAND:  ACALL READY           ;is LCD ready?
          MOV   P1,A            ;issue command code
          CLR   P2.0            ;RS=0 for command
          CLR   P2.1            ;R/W=0 to write to LCD
          SETB  P2.2            ;E=1 for H-to-L pulse
          CLR   P2.2            ;E=0 ,latch in
          RET
```

Program 12-2: Communicating with LCD using the busy flag *(continued on next page)*

```
DATA_DISPLAY:
         ACALL  READY            ;is LCD ready?
         MOV    P1,A             ;issue data
         SETB   P2.0             ;RS=1 for data
         CLR    P2.1             ;R/W=0 to write to LCD
         SETB   P2.2             ;E=1 for H-to-L pulse
         ACALL  DELAY            ;give LCD some time
         CLR    P2.2             ;E=0, latch in
         RET
READY:   SETB   P1.7             ;make P1.7 input port
         CLR    P2.0             ;RS=0 access command reg
         SETB   P2.1             ;R/W=1 read command reg
;read command reg and check busy flag
BACK:    CLR    P2.2             ;E=0 for L-to-H pulse
         ACALL  DELAY            ;give LCD some time
         SETB   P2.2             ;E=1 L-to-H pulse
         JB     P1.7,BACK        ;stay until busy flag=0
         RET
         END
```

Program 12-2. *(continued from previous page)*

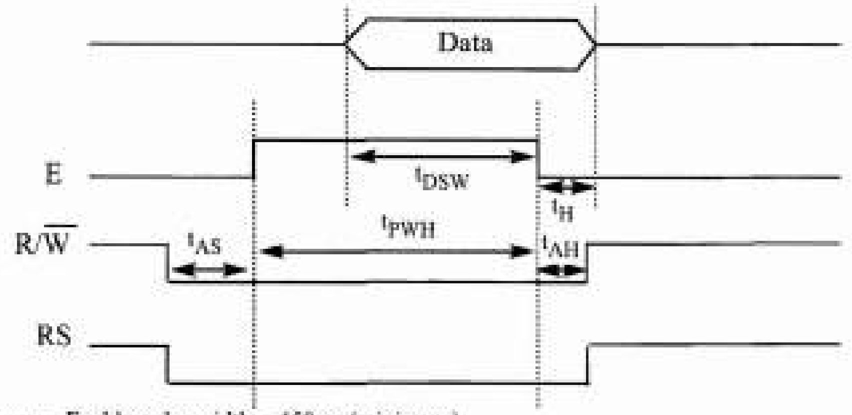
Notice in the above program that the busy flag is D7 of the command register. To read the command register we make R/W = 1 and RS = 0, and a L-to-H pulse for the E pin will provide us the command register. After reading the command register, if bit D7 (the busy flag) is high, the LCD is busy and no information (command or data) should be issued to it. Only when D7 = 0 can we send data or commands to the LCD. Notice in this method that no time delays are used since we are checking the busy flag before issuing commands or data to the LCD. Contrast the Read and Write timing for the LCD in Figures 12-3 and 12-4. Note that the E line is negative-edge triggered for the write while it is positive-edge triggered for the read.



t_D = Data output delay time
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 140 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Note: Read requires an L-to-H pulse for the E pin.

Figure 12-3. LCD Timing for Read (L-to-H for E line)



t_{PWH} = Enable pulse width = 450 ns (minimum)
t_{DSW} = Data setup time = 195 ns (minimum)
t_H = Data hold time = 10 ns (minimum)
t_{AS} = Setup time prior to E (going high) for both RS and R/W = 140 ns (minimum)
t_{AH} = Hold time after E has come down for both RS and R/W = 10 ns (minimum)

Figure 12-4. LCD Timing for Write (H-to-L for E line)

LCD data sheet

In the LCD, one can put data at any location. The following shows address locations and how they are accessed.

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	A	A	A	A	A	A	A

where AAAAAAA = 0000000 to 0100111 for line 1 and AAAAAAA = 1000000 to 1100111 for line 2. See Table 12-3.

Table 12-3: LCD Addressing

	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Line 1 (min)	1	0	0	0	0	0	0	0
Line 1 (max)	1	0	1	0	0	1	1	1
Line 2 (min)	1	1	0	0	0	0	0	0
Line 2 (max)	1	1	1	0	0	1	1	1

The upper address range can go as high as 0100111 for the 40-character-wide LCD, while for the 20-character-wide LCD it goes up to 010011 (19 decimal = 10011 binary). Notice that the upper range 0100111 (binary) = 39 decimal, which corresponds to locations 0 to 39 for the LCDs of 40x2 size.

16 x 2 LCD	80	81	82	83	84	85	86 through 8F
	C0	C1	C2	C3	C4	C5	C6 through CF
20 x 1 LCD	80	81	82	83	through 93		
20 x 2 LCD	80	81	82	83	through 93		
	C0	C1	C2	C3	through D3		
20 x 4 LCD	80	81	82	83	through 93		
	C0	C1	C2	C3	through D3		
	94	95	96	97	through A7		
	D4	D5	D6	D7	through E7		
40 x 2 LCD	80	81	82	83	through A7		
	C0	C1	C2	C3	through E7		

Note: All data is in hex.

Figure 12-5. Cursor Addresses for Some LCDs

From the above discussion we can get the addresses of cursor positions for various sizes of LCDs. See Figure 12-5 for the cursor addresses for common types of LCDs. Note that all the addresses are in hex. Table 12-4 provides a detailed list of LCD commands and instructions. Table 12-2 is extracted from this table.

Optrex is one of the largest manufacturer of LCDs. You can obtain datasheets from their Web site, www.optrex.com.

The LCDs can be purchased from the following Web sites:

www.digikey.com
www.jameco.com
www.elexp.com

Table 12-4: List of LCD Instructions

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (Max)
Clear Display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DD RAM address 0 in address counter	1.64 ms
Return Home	0	0	0	0	0	0	0	0	1		Sets DD RAM address 0 as address counter. Also returns display being shifted to original position. DD RAM contents remain unchanged	1.64 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies shift of display. These operations are performed during data write and read.	40 µs
Display On/ Off Control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of entire display (D), cursor On/Off (C), and blink of cursor position character (B).	40 µs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L			Moves cursor and shifts display without changing DD RAM contents.	40 µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Sets interface data length (DL), number of display lines (L), and character font (F).	40 µs
Set CG RAM Address	0	0	0	1			AGC				Sets CG RAM address. CG RAM data is sent and received after this setting.	40 µs
Set DD RAM Address	0	0	1				ADD				Sets DD RAM address. DD RAM data is sent and received after this setting	40 µs
Read Busy Flag & Address	0	1	BF				AC				Reads Busy flag (BF) indicating internal operation is being performed and reads address counter contents.	40 µs
Write Data CG or DD RAM	1	0				Write Data					Writes data into DD or CG RAM.	40 µs
Read Data CG or DD RAM	1	1				Read Data					Reads data from DD or CG RAM.	40 µs

Notes:

1. Execution times are maximum times when fcp or fosc is 250 kHz.
2. Execution time changes when frequency changes. Ex: When fcp or fosc is 270 kHz: 40 µs × 250 / 270 = 37 µs.
3. Abbreviations:

DD RAM	Display data RAM	
CG RAM	Character generator RAM	
ACC	CG RAM address	
ADD	DD RAM address, corresponds to cursor address	
AC	Address counter used for both DD and CG RAM addresses.	
I/D = 1	Increment	I/D = 0 Decrement
S = 1	Accompanies display shift	
S/C = 1	Display shift,	S/C = 0 Cursor move
R/L = 1	Shift to the right,	R/L = 0 Shift to the left
DL = 1	8 bits, DL = 0: 4 bits	
N = 1	1 line, N = 0 : 1 line	
F = 1	5 x 10 dots, F = 0 : 5 x 7 dots	
BF = 1	Internal operation,	BF = 0 Can accept instruction

Sending information to LCD using MOVC instruction

The Program 12-3 shows how to use the MOVC instruction to send data and commands to an LCD. For an 8051 C version of LCD programming see Examples 12-1 and 12-2.

```
;calls a time delay before sending next data/command
; P1.0-P1.7=D0-D7, P2.0=RS, P2.1=R/W, P2.2=E pins
          ORG   0
          MOV   DPTR,#MYCOM
C1:       CLR   A
          MOVC  A,@A+DPTR
          ACALL COMNWRT       ;call command subroutine
          ACALL DELAY         ;give LCD some time
          JZ    SEND_DAT
          INC   DPTR
          SJMP  C1
SEND_DAT: MOV   DPTR,#MYDATA
D1:       CLR   A
          MOVC  A,@A+DPTR
          ACALL DATAWRT       ;call command subroutine
          ACALL DELAY         ;give LCD some time
          INC   DPTR
          JZ    AGAIN
          SJMP  D1
AGAIN:    SJMP  AGAIN         ;stay here
COMNWRT:                      ;send command to LCD
          MOV   P1,A          ;SEND COMND to P1
          CLR   P2.0          ;RS=0 for command
          CLR   P2.1          ;R/W=0 for write
          SETB  P2.2          ;E=1 for high pulse
          ACALL DELAY         ;give LCD some time
          CLR   P2.2          ;E=0 for H-to-L
          RET
DATAWRT:
          MOV   P1,A          ;SEND DATA to P1
          SETB  P2.0          ;RS=1 for data
          CLR   P2.1          ;R/W=0 for write
          SETB  P2.2          ;E=1 for high pulse
          ACALL DELAY         ;give LCD some time
          CLR   P2.2          ;E=0 for H-to-L pulse
          RET
DELAY:    MOV   R3,#250       ;LONG DELAY FOR fast CPUs
HERE2:    MOV   R4,#255       ;
HERE:     DJNZ  R4,HERE       ;
          DJNZ  R3,HERE2
          RET
          ORG   300H
MYCOM:    DB    38H,0EH,01,06,84H,0     ;commands and null
MYDATA:   DB    "HELLO",0               ;data and null
          END
```

Program 12-3: Sending information to LCD with MOVC instruction.

Example 12-1

Write an 8051 C program to send letters 'M', 'D', and 'E' to the LCD using delays.

Solution:

```
#include <reg51.h>
sfr ldata = 0x90;                    //P1=LCD data pins (Fig. 12-2)
sbit rs = P2^0;
sbit rw = P2^1;
sbit en = P2^2;
void main()
  {
    lcdcmd(0x38);
    MSDelay(250);
    lcdcmd(0x0E);
    MSDelay(250);
    lcdcmd(0x01);
    MSDelay(250);
    lcdcmd(0x06);
    MSDelay(250);
    lcdcmd(0x86);                    //line 1, position 6
    MSDelay(250);
    lcddata('M');
    MSDelay(250);
    lcddata('D');
    MSDelay(250);
    lcddata('E');
  }

void lcdcmd(unsigned char value)
  {
    ldata = value;                   // put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                          // strobe the enable pin
    MSDelay(1);
    en = 0;
    return;
  }

void lcddata(unsigned char value)
  {
    ldata = value;                   // put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                          // strobe the enable pin
    MSDelay(1);
    en = 0;
    return;
  }

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

Example 12-2

Repeat Example 12-1 using the busy flag method.

Solution:

```
#include <reg51.h>
sfr ldata = 0x90;                   //P1=LCD data pins (Fig. 12-2)
sbit rs = P2^0;
sbit rw = P2^1;
sbit en = P2^2;
sbit busy = P1^7;
void main()
  {
    lcdcmd(0x38);
    lcdcmd(0x0E);
    lcdcmd(0x01);
    lcdcmd(0x06);
    lcdcmd(0x86);                   //line 1, position 6
    lcddata('M');
    lcddata('D');
    lcddata('E');
  }

void lcdcmd(unsigned char value)
  {
    lcdready();                     //check the LCD busy flag
    ldata = value;                  //put the value on the pins
    rs = 0;
    rw = 0;
    en = 1;                         //strobe the enable pin
    MSDelay(1);
    en = 0;
    return;
  }

void lcddata(unsigned char value)
  {
    lcdready();                     //check the LCD busy flag
    ldata = value;                  //put the value on the pins
    rs = 1;
    rw = 0;
    en = 1;                         //strobe the enable pin
    MSDelay(1);
    en = 0;
    return;
  }

void lcdready()
  {
    busy = 1;                       //make the busy pin an input
    rs = 0;
    rw = 1;
    while(busy==1)                  //wait here for busy flag
      {
        en = 0;                     //strobe the enable pin
        MSDelay(1);
        en = 1;
      }
    return;
  }
```

Example 12-2 Cont.

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

Review Questions

1. The RS pin is an _______ (input, output) pin for the LCD.
2. The E pin is an ________ (input, output) pin for the LCD.
3. The E pin requires an _______ (H-to-L, L-to-H) pulse to latch in information at the data pins of the LCD.
4. For the LCD to recognize information at the data pins as data, RS must be set to _____ (high, low).
5. Give the command codes for line 1, first character, and line 2, first character.

SECTION 12.2: KEYBOARD INTERFACING

Keyboards and LCDs are the most widely used input/output devices of the 8051, and a basic understanding of them is essential. In this section, we first discuss keyboard fundamentals, along with key press and key detection mechanisms. Then we show how a keyboard is interfaced to an 8051.

Interfacing the keyboard to the 8051

At the lowest level, keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns. In IBM PC keyboards, a single microcontroller (consisting of a microprocessor, RAM and EPROM, and several ports all on a single chip) takes care of hardware and software interfacing of the keyboard. In such systems, it is the function of programs stored in the EPROM of the microcontroller to scan the keys continuously, identify which one has been activated, and present it to the motherboard. In this section we look at the mechanism by which the 8051 scans and identifies the key.

Scanning and identifying the key

Figure 12-6 shows a 4 x 4 matrix connected to two ports. The rows are connected to an output port and the columns are connected to an input port. If no key has been pressed, reading the input port will yield 1s for all columns since they are all connected to high (V_{CC}). If all the rows are grounded and a key is pressed, one of the columns will have 0 since the key pressed provides the path to ground. It is the function of the microcontroller to scan the keyboard continuously to detect and identify the key pressed. How it is done is explained next.

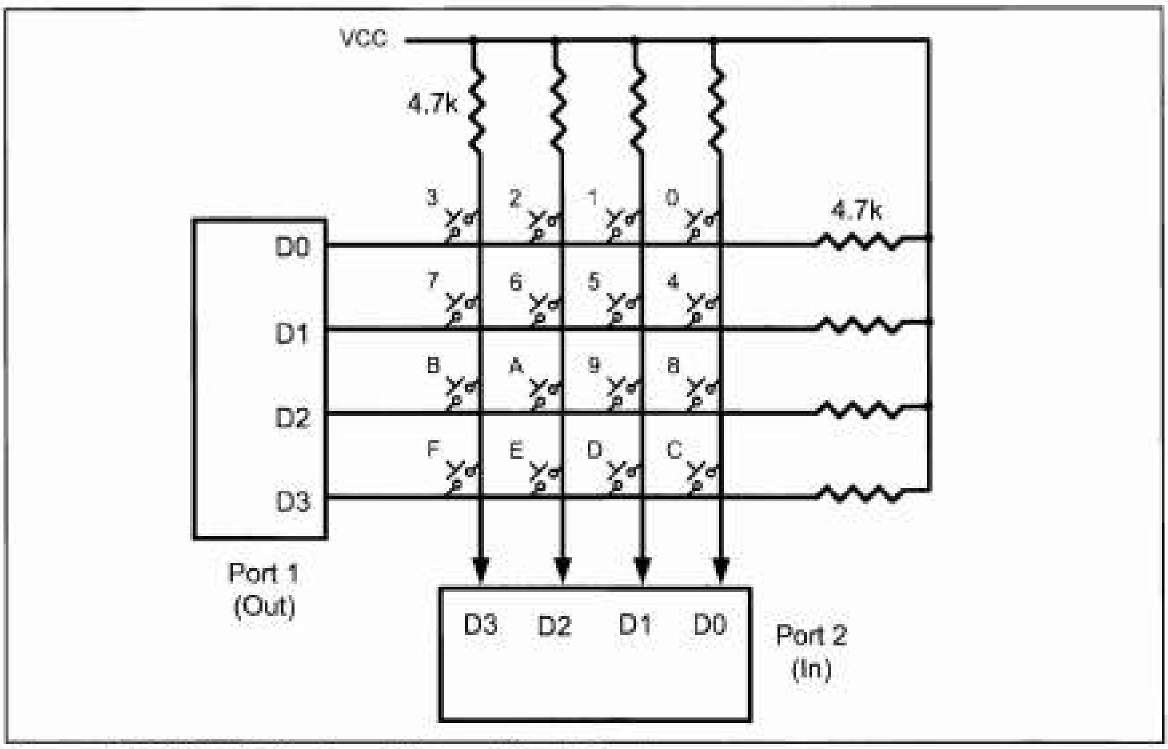


Figure 12-6. Matrix Keyboard Connection to Ports

Grounding rows and reading the columns

To detect a pressed key, the microcontroller grounds all rows by providing 0 to the output latch, then it reads the columns. If the data read from the columns is D3 - D0 = 1111, no key has been pressed and the process continues until a key press is detected. However, if one of the column bits has a zero, this means that a key press has occurred. For example, if D3 - D0 = 1101, this means that a key in the D1 column has been pressed. After a key press is detected, the microcontroller will go through the process of identifying the key. Starting with the top row, the microcontroller grounds it by providing a low to row D0 only; then it reads the columns. If the data read is all 1s, no key in that row is activated and the process is moved to the next row. It grounds the next row, reads the columns, and checks for any zero. This process continues until the row is identified. After identification

Example 12-3

From Figure 12-6, identify the row and column of the pressed key for each of the following.

(a) D3 - D0 = 1110 for the row, D3 - D0 = 1011 for the column

(b) D3 - D0 = 1101 for the row, D3 - D0 = 0111 for the column

Solution:

From Figure 12-6 the row and column can be used to identify the key.

(a) The row belongs to D0 and the column belongs to D2; therefore, key number 2 was pressed.

(b) The row belongs to D1 and the column belongs to D3; therefore, key number 7 was pressed.

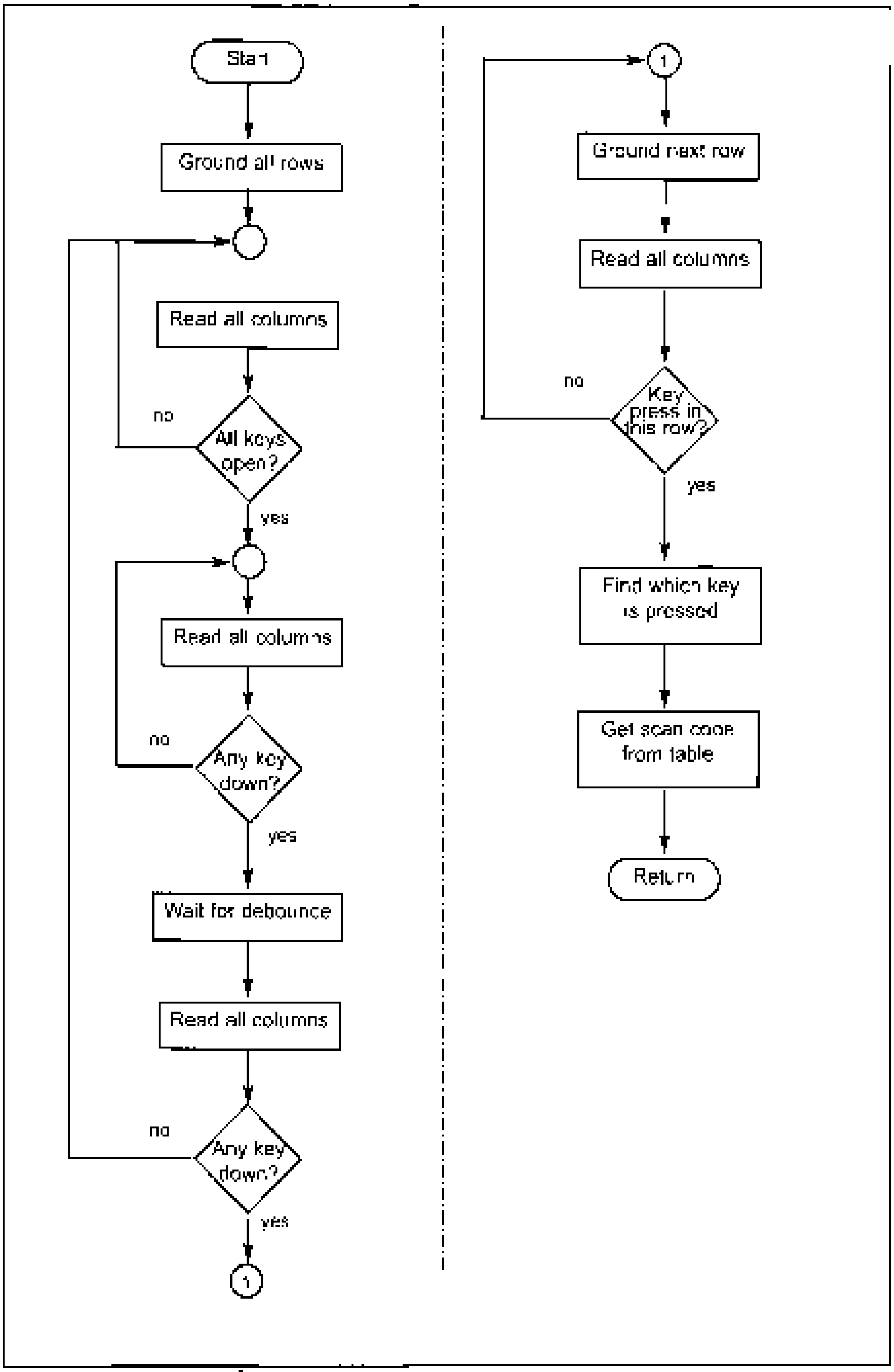


Figure 12-7. Flowchart for Program 12-4

of the row in which the key has been pressed, the next task is to find out which column the pressed key belongs to. This should be easy since the microcontroller knows at any time which row and column are being accessed. Look at Example 12-3.

Program 12-4 is the 8051 Assembly language program for detection and identification of key activation. In this program, it is assumed that P1 and P2 are initialized as output and input, respectively. Program 12-4 goes through the following four major stages:

1. To make sure that the preceding key has been released, 0s are output to all rows at once, and the columns are read and checked repeatedly until all the columns are high. When all columns are found to be high, the program waits for a short amount of time before it goes to the next stage of waiting for a key to be pressed.
2. To see if any key is pressed, the columns are scanned over and over in an infinite loop until one of them has a 0 on it. Remember that the output latches connected to rows still have their initial zeros (provided in stage 1), making them grounded. After the key press detection, the microcontroller waits 20 ms for the bounce and then scans the columns again. This serves two functions: (a) it ensures that the first key press detection was not an erroneous one due to a spike noise, and (b) the 20-ms delay prevents the same key press from being interpreted as a multiple key press. If after the 20-ms delay the key is still pressed, it goes to the next stage to detect which row it belongs to; otherwise, it goes back into the loop to detect a real key press.
3. To detect which row the key press belongs to, the microcontroller grounds one row at a time, reading the columns each time. If it finds that all columns are high, this means that the key press cannot belong to that row; therefore, it grounds the next row and continues until it finds the row the key press belongs to. Upon finding the row that the key press belongs to, it sets up the starting address for the look-up table holding the scan codes (or the ASCII value) for that row and goes to the next stage to identify the key.
4. To identify the key press, the microcontroller rotates the column bits, one bit at a time, into the carry flag and checks to see if it is low. Upon finding the zero, it pulls out the ASCII code for that key from the look-up table; otherwise, it increments the pointer to point to the next element of the look-up table. Figure 12-7 flowcharts this process.

While the key press detection is standard for all keyboards, the process for determining which key is pressed varies. The look-up table method shown in Program 12-4 can be modified to work with any matrix up to 8 x 8. Figure 12-7 provides the flowchart for Program 12-4 for scanning and identifying the pressed key.

There are IC chips such as National Semiconductor's MM74C923 that incorporate keyboard scanning and decoding all in one chip. Such chips use combinations of counters and logic gates (no microcontroller) to implement the underlying concepts presented in Program 12-4. Example 12-4 shows keypad programming in 8051 C.

```
;Keyboard subroutine. This program sends the ASCII code
;for pressed key to P0.1
;P1.0-P1.3 connected to rows P2.0-P2.3 connected to columns
        MOV    P2,#0FFH            ;make P2 an input port
K1:     MOV    P1,#0               ;ground all rows at once
        MOV    A,P2          ;read all col. ensure all keys open
        ANL    A,#00001111B        ;masked unused bits
        CJNE   A,#00001111B,K1     ;check til all keys released
K2:     ACALL  DELAY               ;call 20 ms delay
        MOV    A,P2                ;see if any key is pressed
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,OVER   ;key pressed, await closure
        SJMP   K2                  ;check if key pressed
OVER:   ACALL  DELAY               ;wait 20 ms debounce time
        MOV    A,P2                ;check key closure
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,OVER1  ;key pressed, find row
        SJMP   K2                  ;if none, keep polling
OVER1:  MOV    P1,#11111110B       ;ground row 0
        MOV    A,P2                ;read all columns
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,ROW_0  ;key row 0, find the col.
        MOV    P1,#11111101B       ;ground row 1
        MOV    A,P2                ;read all columns
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,ROW_1  ;key row 1, find the col.
        MOV    P1,#11111011B       ;ground row 2
        MOV    A,P2                ;read all columns
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,ROW_2  ;key row 2, find the col.
        MOV    P1,#11110111B       ;ground row 3
        MOV    A,P2                ;read all columns
        ANL    A,#00001111B        ;mask unused bits
        CJNE   A,#00001111B,ROW_3  ;key row 3, find the col.
        LJMP   K2                  ;if none, false input, repeat

ROW_0:  MOV    DPTR,#KCODE0        ;set DPTR=start of row 0
        SJMP   FIND                ;find col. key belongs to
ROW_1:  MOV    DPTR,#KCODE1        ;set DPTR=start of row 1
        SJMP   FIND                ;find col. key belongs to
ROW_2:  MOV    DPTR,#KCODE2        ;set DPTR=start of row 2
        SJMP   FIND                ;find col. key belongs to
ROW_3:  MOV    DPTR,#KCODE3        ;set DPTR=start of row 3
FIND:   RRC    A                   ;see if any CY bit is low
        JNC    MATCH               ;if zero, get the ASCII code
        INC    DPTR                ;point to next col. address
```

Program 12-4: Keyboard Program *(continued on next page)*

```
        SJMP   FIND                 ;keep searching
MATCH:  CLR    A                    ;set A=0 (match is found)
        MOVC   A,@A+DPTR            ;get ASCII code from table
        MOV    P0,A                 ;display pressed key
        LJMP   K1
;ASCII LOOK-UP TABLE FOR EACH ROW
        ORG    300H
KCODE0: DB     '0','1','2','3'             ;ROW 0
KCODE1: DB     '4','5','6','7'             ;ROW 1
KCODE2: DB     '8','9','A','B'             ;ROW 2
KCODE3: DB     'C','D','E','F'             ;ROW 3
       END
```

Program 12-4. *(continued from previous page)*

Example 12-4

Write a C program to read the keypad and send the result to the first serial port.
P1.0-P1.3 connected to rows
P2.0-P1.3 connected to columns
Configure the serial port for 9600 baud, 8-bit, and 1 stop bit.

Solution:

```
#include <reg51.h>

#define COL  P2             //define ports for easier reading
#define ROW  P1

void MSDelay(unsigned int value);
void SerTX(unsigned char);

unsigned char keypad[4][4] =    {'0','1','2','3',
                                 '4','5','6','7',
                                 '8','9','A','B',
                                 'C','D','E','F'};

void main()
  {
    unsigned char colloc, rowloc;

    TMOD = 0x20;                //timer 1, mode 2
    TH1 = -3;                   //9600 baud
    SCON = 0x50;                //8-bit, 1 stop bit
    TR1 = 1;                    //start timer 1

  //keyboard routine. This sends the ASCII
  //code for pressed key to the serial port
  COL = 0xFF;                   //make P2 an input port
  while(1)                      //repeat forever
    {
    do
      {
      ROW = 0x00;               //ground all rows at once
      colloc = COL;             //read the columns
      colloc &= 0x0F;           //mask used bits
```

Example 12-4 Cont.

```
    } while(colloc != 0x0F);  //check until all keys released

  do
    {
    do
      {
      MSDelay(20);                //call delay
      colloc = COL;               //see if any key is pressed
      colloc &= 0x0F;             //mask unused bits
      } while(colloc == 0x0F);//keep checking for keypress

    MSDelay(20);                  //call delay for debounce
    colloc = COL;                 //read columns
    colloc &= 0x0F;               //mask unused bits
    } while(colloc == 0x0F);      //wait for keypress

    while(1)
      {
        ROW = 0xFE;               //ground row 0
        colloc = COL;             //read columns
        colloc &= 0x0F;           //mask unused bits
        if(colloc != 0x0F)        //column detected
          {
            rowloc = 0;           //save row location
            break;                //exit while loop
          }

        ROW = 0xFD;               //ground row 1
        colloc = COL;             //read columns
        colloc &= 0x0F;           //mask unused bits
        if(colloc != 0x0F)        //column detected
          {
            rowloc = 1;           //save row location
            break;                //exit while loop
          }

        ROW = 0xFB;               //ground row 2
        colloc = COL;             //read columns
        colloc &= 0x0F;           //mask unused bits
        if(colloc != 0x0F)        //column detected
          {
            rowloc = 2;           //save row location
            break;                //exit while loop
          }

        ROW = 0xF7;               //ground row 3
        colloc = COL;             //read columns
        colloc &= 0x0F;           //mask unused bits
        rowloc = 3;               //save row location
        break;                    //exit while loop
      }

    //check column and send result to the serial port
    if(colloc == 0x0E)
      SerTX(keypad[rowloc][0]);
    else if(colloc == 0x0D)
      SerTX(keypad[rowloc][1]);
    else if(colloc == 0x0B)
      SerTX(keypad[rowloc][2]);
```

Example 12-4 Cont.

```
        else
          SerTX(keypad[rowloc][3]);
        }

   }

void SerTX(unsigned char x)
   {
     SBUF = x;                //place value in buffer
     while(TI==0);            //wait until transmitted
     TI = 0;                  //clear flag
   }

void MSDelay(unsigned int value)
   {
     unsigned int x, y;
     for(x=0;x<1275;x++)
       for(y=0;y<value;y++);
   }
```

Review Questions

1. True or false. To see if any key is pressed, all rows are grounded.
2. If D3 - D0 = 0111 is the data read from the columns, which column does the pressed key belong to?
3. True or false. Key press detection and key identification require two different processes.
4. In Figure 12-6, if the rows are D3 - D0 = 1110 and the columns are D3 - D0 = 1110, which key is pressed?
5. True or false. To identify the pressed key, one row at a time is grounded.

SUMMARY

This chapter showed how to interface real-world devices such as LCDs and keypads to the 8051. First, we described the operation modes of LCDs, then described how to program the LCD by sending data or commands to it via its interface to the 8051.

Keyboards are one of the most widely used input devices for 8051 projects. This chapter also described the operation of keyboards, including key press and detection mechanisms. Then the 8051 was shown interfacing with a keyboard. 8051 programs were written to return the ASCII code for the pressed key.

PROBLEMS

SECTION 12.1: LCD INTERFACING

1. The LCD discussed in this section has _______ (4, 8) data pins.
2. Describe the function of pins E, R/W, and RS in the LCD.
3. What is the difference between the V_{CC} and V_{EE} pins on the LCD?
4. "Clear LCD" is a __________ (command code, data item) and its value is ___ hex.
5. What is the hex value of the command code for "display on, cursor on"?
6. Give the state of RS, E, and R/W when sending a command code to the LCD.
7. Give the state of RS, E, and R/W when sending data character 'Z' to the LCD.
8. Which of the following is needed on the E pin in order for a command code (or data) to be latched in by the LCD?
   (a) H-to-L pulse (b) L-to-H pulse
9. True or false. For the above to work, the value of the command code (data) must already be at the D0 - D7 pins.
10. There are two methods of sending streams of characters to the LCD: (1) checking the busy flag, or (2) putting some time delay between sending each character without checking the busy flag. Explain the difference and the advantages and disadvantages of each method. Also explain how we monitor the busy flag.
11. For a 16x2 LCD, the location of the last character of line 1 is 8FH (its command code). Show how this value was calculated.
12. For a 16x2 LCD, the location of the first character of line 2 is C0H (its command code). Show how this value was calculated.
13. For a 20x2 LCD, the location of the last character of line 2 is 93H (its command code). Show how this value was calculated.
14. For a 20x2 LCD, the location of the third character of line 2 is C2H (its command code). Show how this value was calculated.
15. For a 40x2 LCD, the location of the last character of line 1 is A7H (its command code). Show how this value was calculated.
16. For a 40x2 LCD, the location of the last character of line 2 is E7H (its command code). Show how this value was calculated.
17. Show the value (in hex) for the command code for the 10th location, line 1 on a 20x2 LCD. Show how you got your value.
18. Show the value (in hex) for the command code for the 20th location, line 2 on a 40x2 LCD. Show how you got your value.
19. Rewrite the COMNWRT subroutine. Assume connections P1.4 = RS, P1.5 = R/W, P1.6 = E.
20. Repeat Problem 19 for the data write subroutine. Send the string "Hello" to the LCD by checking the busy flag. Use the instruction MOVC.

SECTION 12.2: KEYBOARD INTERFACING

21. In reading the columns of a keyboard matrix, if no key is pressed we should get all ______ (1s, 0s).

22. In Figure 12-6, to detect the key press, which of the following is grounded?
    (a) all rows (b) one row at time (c) both (a) and (b)
23. In Figure 12-6, to identify the key pressed, which of the following is grounded?
    (a) all rows (b) one row at time (c) both (a) and (b)
24. For Figure 12-6, indicate the column and row for each of the following.
    (a) D3 - D0 = 0111 (b) D3 - D0 = 1110
25. Indicate the steps to detect the key press.
26. Indicate the steps to identify the key pressed.
27. Indicate an advantage and a disadvantage of using an IC chip for keyboard scanning and decoding instead of using a microcontroller.
28. What is the best compromise for the answer to Problem 27?

ANSWERS TO REVIEW QUESTIONS

SECTION 12.1: LCD INTERFACING

1. Input
2. Input
3. H-to-L
4. High
5. 80H and C0H

SECTION 12.2: KEYBOARD INTERFACING

1. True
2. Column 3
3. True
4. 0
5. True

QUESTION 1

Develop an 8051 Assembly Language Program (ALP) for the following scenario. The port P1 receives the data from the user. Based on the nature of the received data do the following:
If the received data at P1 is EVEN then send it to P2 by inverting bits that are in the even position.
If the received data at P1 is ODD then send it to P3 by inverting bits that are in the odd position.

Solution

```
ORG 0000H

MOV P1,#0FFH
MOV A,P1
ANL A,#01H
JNZ ODD

MOV A,P1
XRL A,#55H
MOV P2,A
SJMP END1

ODD:
MOV A,P1
XRL A,#0AAH
MOV P3,A

END1:
SJMP END1

END
```

QUESTION 2

Develop an 8051 ALP to interface a capacitive touch sensor with Port P3.4 and use Timer 0 to detect touch events. When the sensor is touched, increment the count on Port P1 and display the touch count on Port P2 every 1 second using Timer 1.

Solution

```
ORG 0000H

MOV TMOD,#11H
MOV P3,#0FFH
MOV P1,#00H

MAIN:
JB P3.4,MAIN

INC P1

MOV TH1,#3CH
MOV TL1,#0B0H
SETB TR1

WAIT1:
JNB TF1,WAIT1

CLR TR1
CLR TF1

MOV A,P1
MOV P2,A

SJMP MAIN

END
```

QUESTION 3

Develop an 8051 ALP to receive data via the serial port at a baud rate of 9600. The received data should be buffered in Port P1, and the program should send an acknowledgment message ("Data Received") back to the sender using interrupt.

Solution

```
ORG 0000H
LJMP MAIN

ORG 0023H
```

```
MOV A,SBUF
MOV P1,A

MOV SBUF,#'D'
ACALL WAIT
MOV SBUF,#'A'
ACALL WAIT
MOV SBUF,#'T'
ACALL WAIT
MOV SBUF,#'A'
ACALL WAIT

RETI

MAIN:

MOV TMOD,#20H
MOV TH1,#0FDH
MOV SCON,#50H

SETB TR1
SETB ES
SETB EA

HERE: SJMP HERE

WAIT:
JNB TI,WAIT
CLR TI
RET

END
```

QUESTION 4

Develop an 8051 ALP to interface 16×2 LCD display. Assume the string “VIT” is stored in ROM location from 200H onwards and the string “Vellore” is stored in ROM location from 300H onwards. A switch is connected to port pin P0.4. If the switch input is 1 display “VIT” in line 1 otherwise display “Vellore” in line 2.

Solution

```
ORG 0000H

MOV P0,#0FFH

CHECK:
JB P0.4,LINE1

MOV DPTR,#0300H
ACALL DISP2
SJMP CHECK

LINE1:
MOV DPTR,#0200H
ACALL DISP1
SJMP CHECK

DISP1:
CLR A
MOVC A,@A+DPTR
JZ EXIT1
MOV P2,A
INC DPTR
SJMP DISP1

EXIT1: RET

DISP2:
CLR A
MOVC A,@A+DPTR
JZ EXIT2
MOV P2,A
INC DPTR
SJMP DISP2

EXIT2: RET

END
```

QUESTION 5

An 8051 microcontroller is connected to a linear temperature sensor through ADC0804 which gives 5V when the temperature is 100°C. Develop an ALP to interface ADC0804 with 8051 and compare the value in accumulator with threshold value 50°C. If A > 50°C turn ON buzzer at P1.3 for 60 ms using Timer 0.

Solution

```
ORG 0000H

MOV TMOD,#01H

START:

CLR P3.6
SETB P3.6

WAIT:
JB P3.2,WAIT

MOV A,P0

MOV R0,#32H
CLR C
SUBB A,R0

JNC ALARM
SJMP START

ALARM:

SETB P1.3

MOV TH0,#0FCH
MOV TL0,#66H
SETB TR0

W1: JNB TF0,W1

CLR TR0
CLR TF0
CLR P1.3

SJMP START

END
```

QUESTION 6

How the physical address is calculated and find it if segment address is 4000H and offset address is 3000H. Why are these two addresses required in 8086?

Solution

Physical Address

PHY ADD = SEG ADD × 10H + OFFSET

SEG ADD = 4000H
OFFSET = 3000H

PHY ADD = 4000H × 10H

= 40000H

PHY ADD = 40000H + 3000H

= 43000H

Reason

8086 has 16-bit registers but uses **20-bit physical address** to access **1 MB memory**. Therefore address is generated using

Segment Address + Offset Address.

QUESTION 7

Find the errors in the following instructions (if any).

i. MOV CS,5000H
ii. ADD [AX],[4000H]
iii. ADD 0200H
iv. INC 3000H
v. CMP BX

Solution

i

MOV CS,5000H → Invalid

Correct

```
MOV AX,5000H
MOV CS,AX
```

ii

Memory to memory not allowed

Correct

```
ADD AX,[4000H]
```

iii

Destination missing

Correct

```
ADD AX,0200H
```

iv

Memory operand must use brackets

Correct

```
INC [3000H]
```

v

Two operands required

Correct

```
CMP AX,BX
```

QUESTION 8

Write an 8086 ALP to find the average of 64 numbers stored in memory using DIV instruction.

Solution

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
NUM DB 64 DUP(?)
RESULT DW ?
DATA ENDS

CODE SEGMENT

START

MOV AX,DATA
MOV DS,AX

MOV CX,40H
MOV SI,5000H
MOV AX,0000H

L1

MOV BL,[SI]
ADD AX,BX
INC SI
DEC CX
JNZ L1

MOV BX,40H
DIV BX

MOV RESULT,AX

HLT

CODE ENDS
END START
```

QUESTION 9

Write an 8086 ALP to find the average of 64 numbers without using DIV instruction.

Solution

```
ASSUME CS:CODE, DS:DATA

DATA SEGMENT
NUM DB 64 DUP(?)
RESULT DW ?
DATA ENDS

CODE SEGMENT

START

MOV AX,DATA
MOV DS,AX

MOV CX,40H
MOV SI,5000H
MOV AX,0000H

L1

MOV BL,[SI]
ADD AX,BX
INC SI
DEC CX
JNZ L1

MOV CL,06H
SHR AX,CL

MOV RESULT,AX

HLT

CODE ENDS
END START
```

QUESTION 10

Write an 8051 ALP to find the numbers divisible by both 3 and 7 in the range 1 to 99 and store them in memory. Display the numbers using two seven segment displays connected to P0 and P1.

Solution

```
ORG 0000H

MOV R0,#20H
MOV R3,#01H

LOOP

MOV A,R3
MOV B,#21
DIV AB

MOV A,B
CJNE A,#00H,NEXT

MOV A,R3
MOV @R0,A
INC R0

NEXT

INC R3
CJNE R3,#64H,LOOP

SJMP $

END
```

QUESTION 11

Determine the number of cycles required for DMA transfer of a file of size 29154 KB if each DMA cycle transfers 2^{16} bytes.

Solution

File size

29154 KB

1 KB = 2^{10} bytes

File size

= 29154 × 2^{10}

= 29853696 bytes

Bytes per cycle

2^{16} = 65536 bytes

Number of cycles

= 29853696 / 65536

= 455.53

Required cycles

= **456 cycles**