MICROPROCESSOR LABORATORY MANUAL



T.C.

AYDIN ADNAN MENDERES

UNIVERSITY ENGINEERING

FACULTY

ELECTRICAL & ELECTRONICS ENGINEERING DEPARTMENT

Lecturers:
Asst. Prof. Münevver Mine ÖZYETKİN, PhD.
Res. Asst. İsmail YARİÇİ (Room C315, ismail.yarici@adu.edu.tr)

Prepared: Res. Asst. Abdullah GÜLDEREN

Aydın – 2021

AYDIN ADNAN MENDERES UNIVERSITY
EE312 – MICROPROCESSOR LABORATORY (2021/2022)

Course Calendar		
# of Exp.	**Date**	**Topic**
1	**07.10.2021**	**-**
2	**14.10.2021**	**Meeting, General Information About The Course**
3	**21.10.2021**	**Keil Mvision And Proteus Software. Create New Project And Compilation Steps.**
4	**28.10.2021**	**Adressing Mods, Choosing Bank, Read and Write Data**
5	**04.11.2021**	**Arithmetic and Logic Instructions**
6	**11.11.2021**	**Branch Operations and Time Delays**
7	**18.11.2021**	**8051 Project with Loops, and Time Delays**
8	**25.11.2021**	**Midterm Week**
9	**02.12.2021**	**Midterm Week**
10	**09.12.2021**	**Using 7 Segment Display with 8051**
11	**16.12.2021**	**8051 Timer and Counters**
12	**23.12.2021**	**The 8051 Interrupts**
13	**30.12.2021**	**Project1 with 8051**
14	**06.01.2022**	**Project2 with 8051**
15	**13.01.2022**	**Make up**

About This Laboratory

1. Students should be prepared for the experiment to be conducted.
2. Everyone should have his own experiment booklet.
3. A calculator should be brought to the experiment.
4. Laboratory works, duties must be done individually.
5. Laboratory attendance requirement is 80%.
6. Students must be in the lab on time.
7. The grading percentage of this course will be as follows.

	Percentage (%)
Attendace	5
Laboratory Performances and reports	60
Projects	35
Total	**100**

INTRODUCTION

A microcontroller is a computer with most of the necessary support chips onboard. All computers have several things in common, namely:

- A central processing unit (CPU) that 'executes' programs.
- Some random-access memory (RAM) where it can store data that is variable.
- Some read only memory (ROM) where programs to be executed can be stored.
- Input and output (I/O) devices that enable communication to be established with the outside world i.e. connection to devices such as keyboard, mouse, monitors and other peripherals.

There are many versions of microcontrollers 8051, AT89C51 from Atmel Corporation and many more. In this manual we will study about the 8051 architecture, its features, programming and interfacing.

MCS 8051 is an 8-bit single chip microcontroller with many built-in functions and is the core for all MCS-51 devices. The main features of the 8051 core are:

- Operates with single Power Supply +5V.
- 8-bit CPU optimized for control applications.
- 16-bit program counter (PC) and 16-bit data pointer (DPTR).
- 8-bit program status word (PSW).
- 8-bit stack pointer (SP).
- 4K Bytes of On-Chip Program Memory (Internal ROM or EPROM).
- 128 bytes of On-Chip Data Memory (Internal RAM):
    - Four Register Banks, each containing 8 registers (R0 to R7) [Total 32 reg]
    - 16-bytes of bit addressable memory.
    - 80 bytes of general-purpose data memory (Scratch Pad Area).
- Special Function Registers (SFR) to configure/operate microcontroller.
- 32 bit bi-directional I/O Lines (4 ports P0 to P3).
- Two 16-bit timers/counters (T0 and T1).
- Full duplex UART (Universal Asynchronous Receiver/Transmitter).
- On-Chip oscillator and clock circuitry.

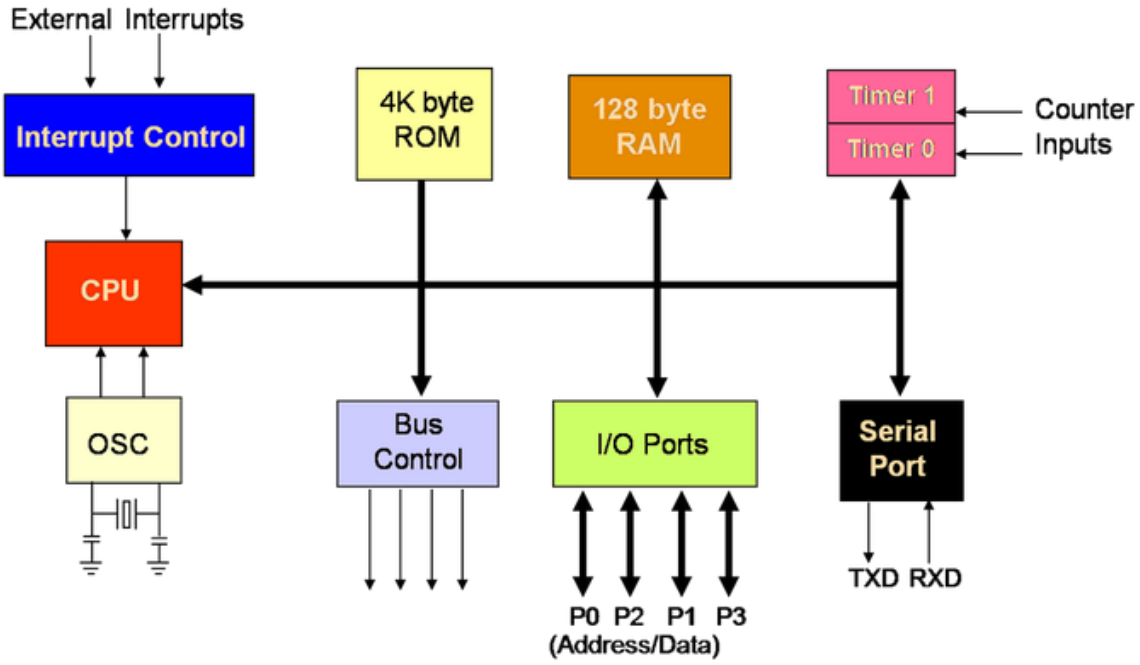


Figure 1.1 General block diagram of 8051 microcontroller architecture [3].

8051 Pinout

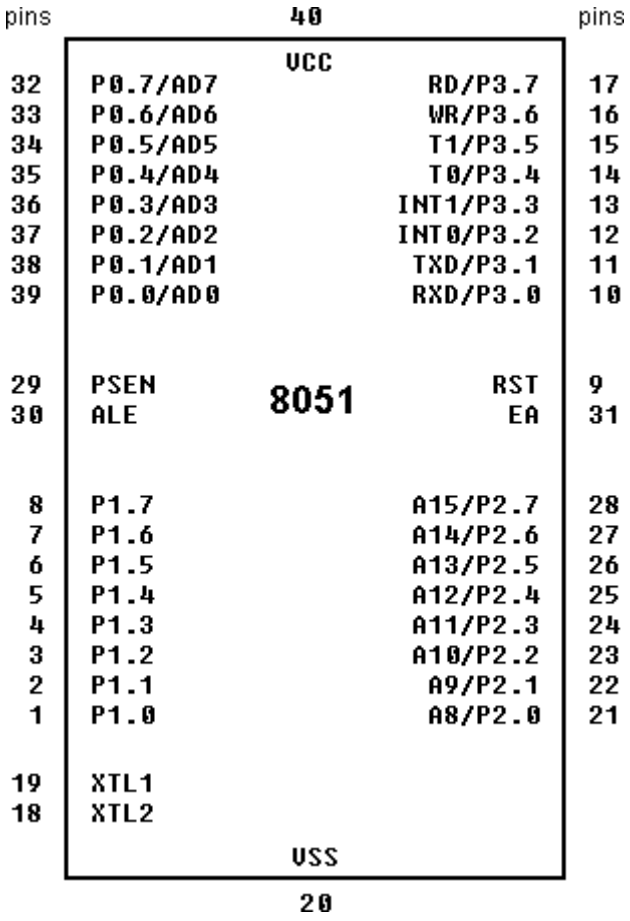


Figure 1.2

The diagram above shows the 8051 pinout. The chip is a 40-pin package.

Port 0 - pins 32 to 39 make up the 8-bit I/O port 0. However, if external memory is used, these lines are used as a multiplexed address and data bus.

Port 1 - pins 1 to 8 make up the 8-bit I/O port 1.

Port 2 - pins 21 to 28 make up the 8-bit I/O port 2. However, if external memory is used, these lines make up the high-byte of the external address (A8 to A15).

Port 3 - pins 10 to 17 make up the 8-bit I/O port 3. However, each of these eight pins also has an alternate function, as detailed in the table below.

Pin	Name	Bit Address	Function
P3.0	RXD	B0H	Receive data for serial port
P3.1	TXD	B1H	Transmit data for serial port
P3.2	INT0-bar	B2H	External interrupt 0
P3.3	INT1-bar	B3H	External interrupt 1
P3.4	T0	B4H	Timer/counter 0 external input
P3.5	T1	B5H	Timer/counter 1 external input
P3.6	WR-bar	B6H	External data memory write strobe
P3.7	RD-bar	B7H	External data memory read strobe

RST - the reset input is on pin 9. This pin is used for resetting the 8051 (ie; loading the PC with the correct startup value).

EA-bar - the external access, on pin 31, is used for enabling or disabling the on-chip ROM. When tied high (5V), the 8051 executes instructions in internal ROM when executing in the lower 4K (8K for the 8052) of memory. If tied low the 8051 will always execute instructions in external memory. The 8031 and 8032 should always have pin 31 tied low as there is no internal code memory.

ALE - the address latch enable is on pin 30. The ALE is used for latching the low byte of the address into an external register. We will deal with this at a later date.

PSEN-bar - the program store enable is an output signal on pin 29. This signal is used for fetching instructions from external code memory. We will also deal with this at a later date.

8051 Memory

Organisation Internal

ROM

The 8051 has 4K (4096 locations) of on-chip ROM. This is used for storing the system program. 2^{12} = 4096, therefore the internal ROM address bus is 12 bits wide and internal ROM locations go from 000H to FFFH.

Internal RAM

There are 256 bytes of internal RAM on the 8051. 2^8 = 256, therefore the internal RAM address bus is 8 bits wide and internal RAM locations go from 00H to FFH.

The first 128 locations (00H to 7FH) of internal RAM are used by the programmer for storing data while the second 128 locations (80H to FFH) are the Special Function Registers (SFRs) which we will deal with later.

The diagram below is a summary of the 8051 on-chip RAM.

Register Banks

There are four register banks from 00H to 1FH. On power-up, registers R0 to R7 are located at 00H to 07H. However, this can be changed so that the register set points to any of the other three banks (if you change to Bank 2, for example, R0 to R7 is now located at 10H to 17H).

Bit-addressable Locations

The 8051 contains 210 bit-addressable locations of which 128 are at locations 20H to 2FH while the rest are in the SFRs. Each of the 128 bits from 20H to 2FH have a unique number (address) attached to them, as shown in the table above. The 8051 instruction set allows you to set or reset any single bit in this section of RAM.

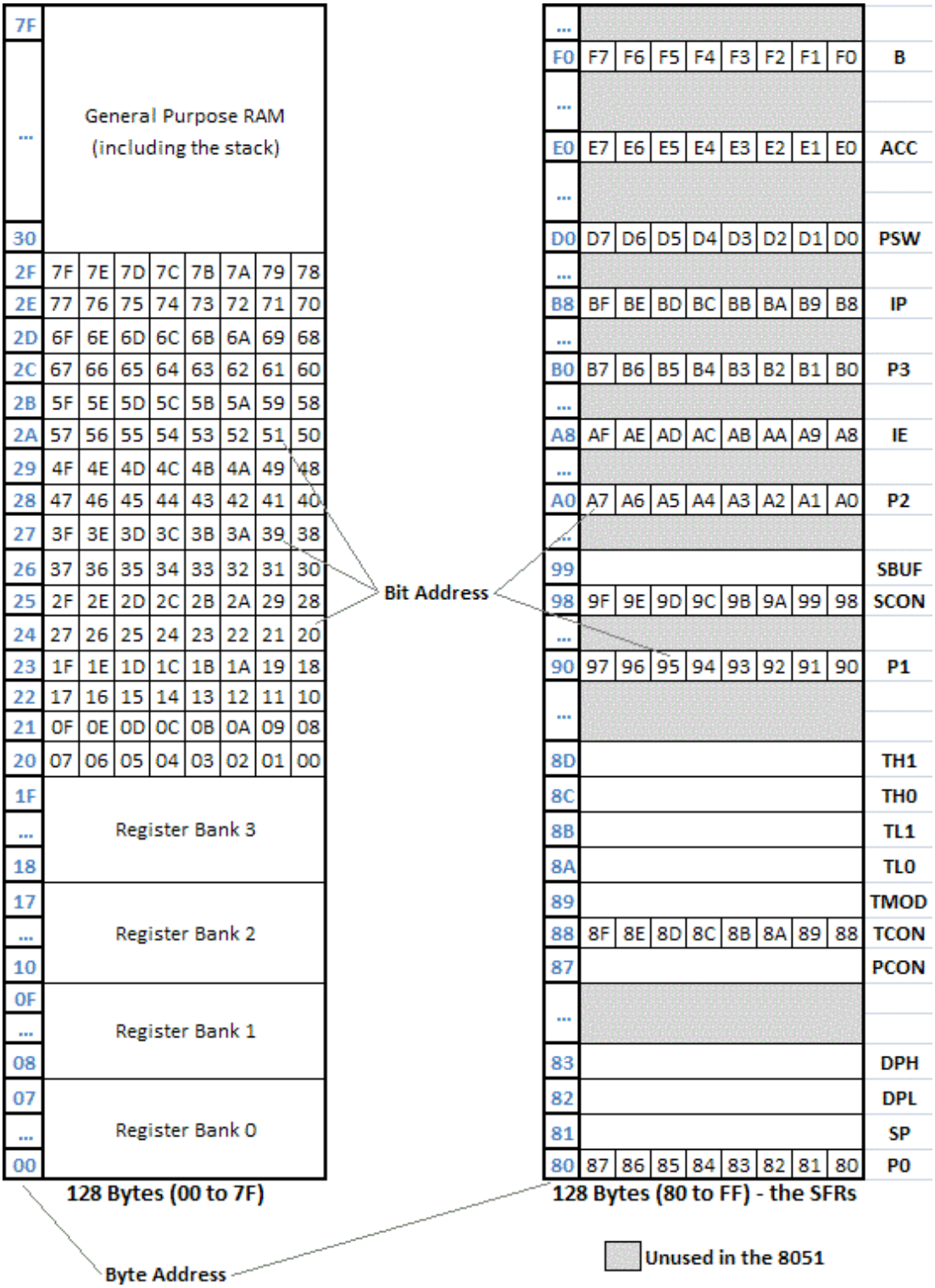
With the general purpose RAM from 30H to 7FH and the register banks from 00H to 1FH, you may only read or write a full byte (8 bits) at these locations.

However, with bit-addressable RAM (20H to 2FH) you can read or write any single bit in this region by using the unique address for that bit. We will later see that this is a very powerful feature.

Special Function Registers (SFRs)

Locations 80H to FFH contain the special function registers. As you can see from the diagram below, not all locations are used by the 8051 (eleven locations are blank). These extra locations are used by other family members (8052, etc.) for the extra features these microcontrollers possess. Also note that not all SFRs are bit-addressable. Those that are have a unique address for each bit.

We will deal with each of the SFRs as we progress through the course, but for the moment you should take note of the accumulator (ACC) at address E0H and the four port registers at addresses 80H for P0, 90h for P1, A0 for P2 and B0 for P3. We will later see how easy this makes reading from and writing to any of the four ports.



The Missing Registers

The two registers not shown in the table above are the instruction register and the program counter. These two very important registers are not placed in RAM because it is not necessary to make them directly accessible to the programmer. The instruction register gets its data from the execution cycle while the program counter is manipulated by the operation of these instructions.

Program Status Word (PSW)

The PSW is at location D0H and is bit addressable. The table below describes the function of each bit.

Bit	Symbol	Address	Description
PSW.7	CY	D7H	Carry flag
PSW.6	AC	D6H	Axuillary carry flag
PSW.5	F0	D5H	Flag 0
PSW.4	RS1	D4H	Register bank select 1
PSW.3	RS0	D3H	Register bank select 0
PSW.2	OV	D2H	Overflow flag
PSW.1	--	D1H	Reserved
PSW.0	P	D0H	Even parity flag

Carry Flag

The carry flag has two functions.

- Firstly, it is used as the carry-out in 8-bit addition/subtraction. For example, if the accumulator contains FDH and we add 3 to the contents of the accumulator (ADD A, #3), the accumulator will then contain zero and the carry flag will be set. It is also set if a subtraction causes a borrow into bit 7. In other words, if a number is subtracted from another number smaller than it, the carry flag will be set. For example, if A contains 3DH and R3 contains 4BH, the instruction SUBB A, R3 will result in the carry bit being set (4BH is greater than 3DH).
- The carry flag is also used during Boolean operations. For example, we could AND the contents of bit 3DH with the carry flag, the result being placed in the carry flag - ANL C, 3DH

Register Bank Select Bit

Bits 3 and 4 of the PSW are used for selecting the register bank. Since there are four register banks, two bits are required for selecting a bank, as detailed below.

PSW.4	PSW.3	Register Bank	Address of Register Bank
0	0	0	00H to 07H
0	1	1	08H to 0FH
1	0	2	10H to 17H
1	1	3	18H to 1FH

For example, if we wished to activate register bank 3 we would use the following instructions- SETB RS1

SETB RS0

If we then moved the contents of R4 to the accumulator (MOV A, R4) we would be moving the data from location 1CH to A.

Flag 0

Flag 0 is a general-purpose flag available to the programmer.

Parity Bit

The parity bit is automatically set or cleared every machine cycle to ensure even parity with the accumulator. The number of 1-bits in the accumulator plus the parity bit is always even. In other words, if the number of 1s in the accumulator is odd then the parity bit is set to make the overall number of bits even. If the number of 1s in the accumulator is even then the parity bit is cleared to make the overall number of bits even.

For example, if the accumulator holds the number 05H, this is 0000 0101 in binary => the accumulator has an even number of 1s, therefore the parity bit is cleared. If the accumulator holds the number F2H, this is 1111 0010 => the accumulator has an odd number of 1s, therefore the parity bit is set to make the overall number of 1s even.

As we shall see later in the course, the parity bit is most often used for detecting errors in transmitted data.

B Register

The B register is used together with the accumulator for multiply and divide operations.

- The MUL AB instruction multiplies the values in A and B and stores the low-byte of the result in A and the high-byte in B.
    - For example, if the accumulator contains F5H and the B register contains 02H, the result of MUL AB will be A = EAH and B = 01H.
- The DIV AB instruction divides A by B leaving the integer result in A and the remainder by B.
    - For example, if the accumulator contains 07H and the B register contains 02H, the result of DIV AB will be A = 03H and B = 01H.

The B register is also bit-addressable.

Stack Pointer

The stack pointer (SP) is an 8-bit register at location 81H. A stack is used for temporarily storing data. It operates on the basis of last in first out (LIFO). Putting data onto the stack is called "pushing onto the stack" while taking data off the stack is called "popping the stack."

The stack pointer contains the address of the item currently on top of the stack. On power- up or reset the SP is set to 07H. When pushing data onto the stack, the SP is first increased by one and the data is then placed in the location pointed to by the SP. When popping the stack, the data is taken off the stack and the SP is then decreased by one.

Since reset initialises the SP to 07H, the first item pushed onto the stack is stored at 08H (remember, the SP is incremented first, then the item is placed on the stack). However, if the programmer wishes to use the register banks 1 to 3, which start at address 08H, he/she must move the stack to another part of memory. The general purpose RAM starting at address 30H is a good spot to place the stack. To do so we need to change the contents of the SP.

MOV SP, #2FH.

Now, the first item to be pushed onto the stack will be stored at 30H.

Steps to Create and Compile Program Using Keil µVision5

1. Download Keil µVision5 and install it on your computer.
2. Double click on the Keil µVision5 icon on the desktop.



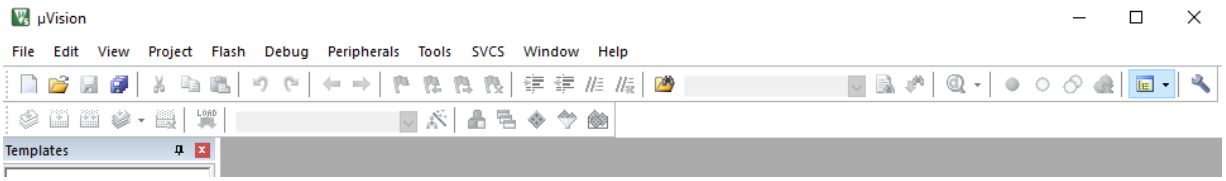


Figure 1.3 Keil µVision5 Simulation window

3. Close any previous projects that were opened using – **Project -> Close**.
4. Start **Project – New µVision Project** and write a file name after click **'Save'**.

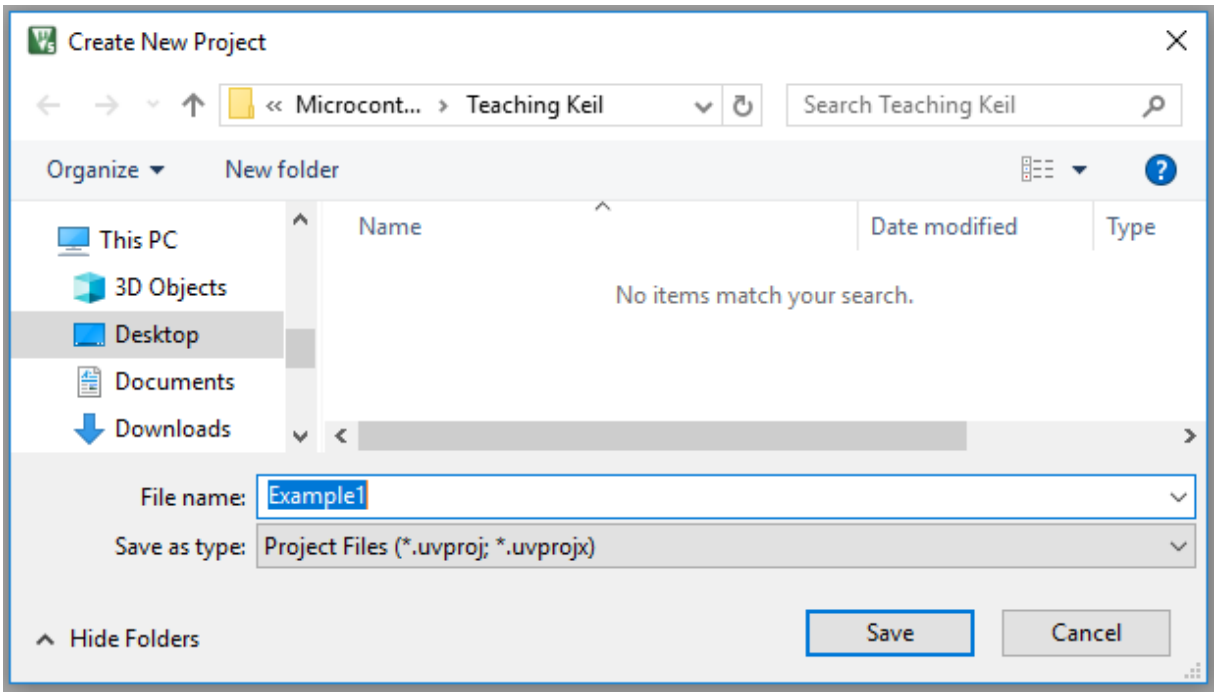


Figure 1.4

5. Select the CPU from the device database. We'll use the AT89C52 CPU here. On clicking **'OK'**, the following option is displayed. Choose **'No'**.

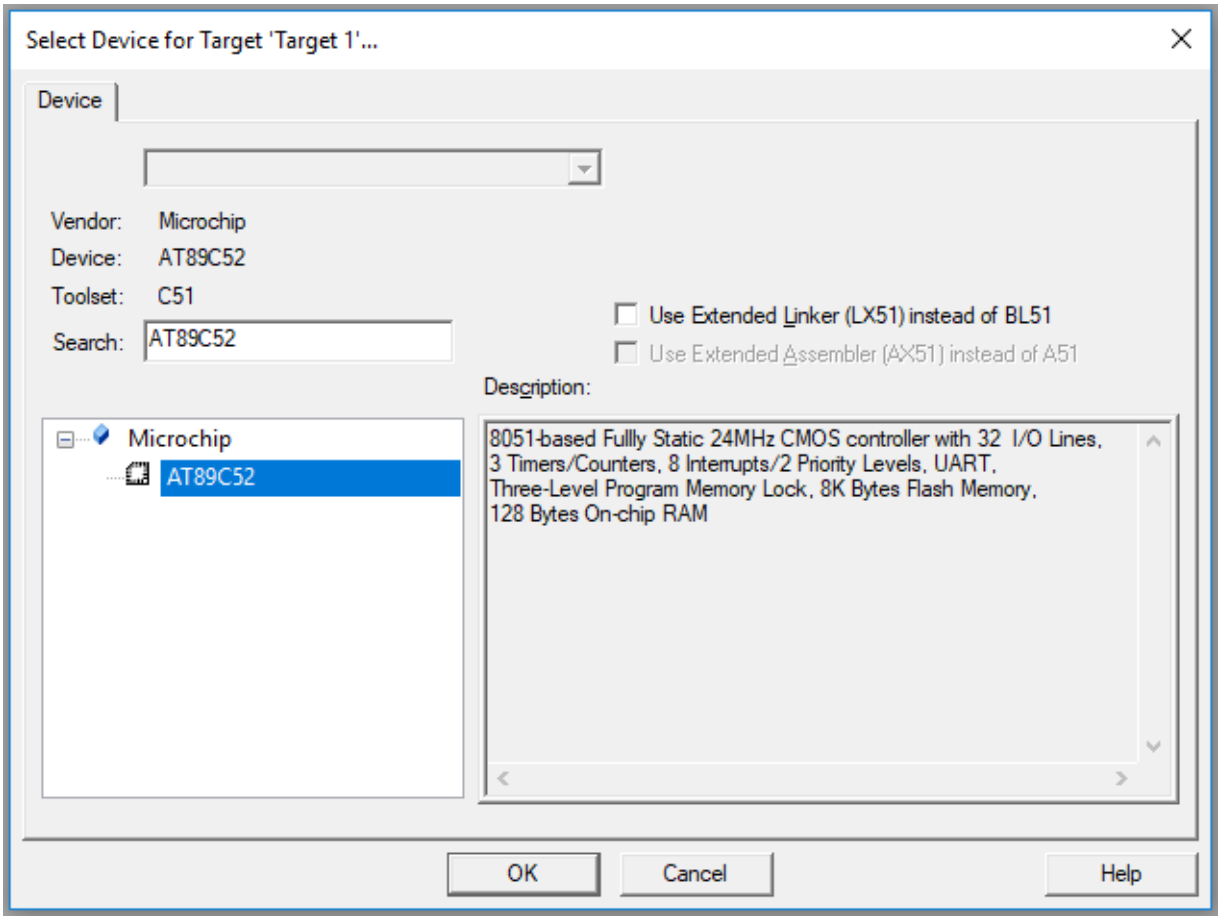


Figure 1.5

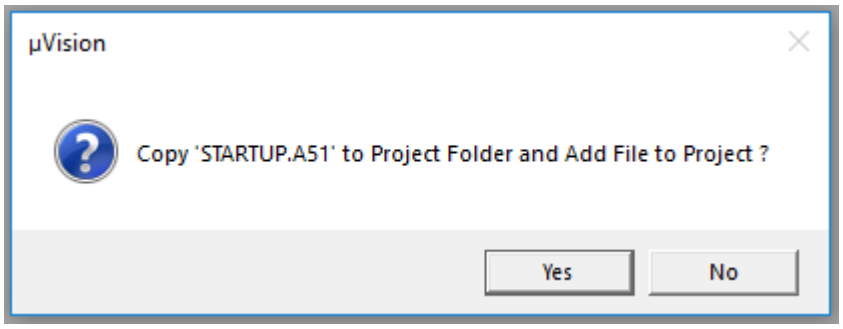


Figure 1.6

6. Create a source file (using **File->New**), type in the assembly or C program and save this (filename**.asm**/filename**.c**)

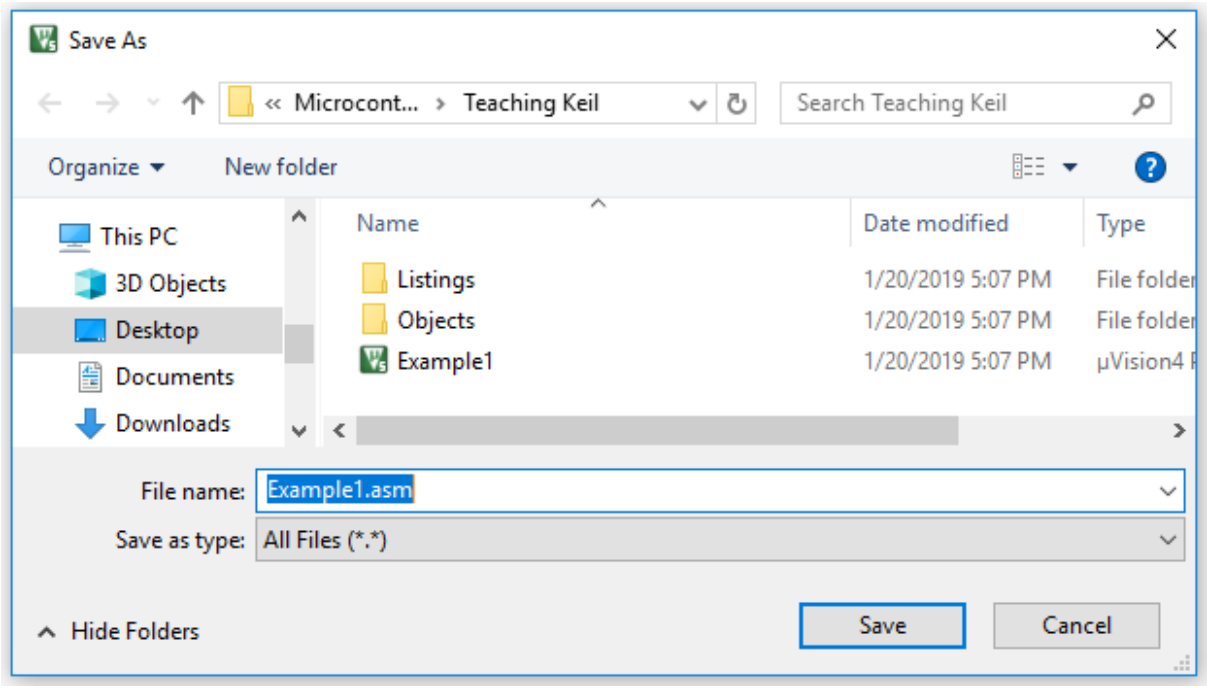


Figure 1.7

7. And add this source file to the project using either one of the following two methods.
   i) **Project**->**Manage**->**Project Items** -> **Addfiles** -> browse to the required file -> **OK** (Figure 1.8)
   ii) Right click on the Source Group in the Project Window and the Add Files to Group option. (Figure 1.9 - 1.10)
8. Set the Target options using -> **Project** – **Options for Target** opens the µVision **Options for Target – 'Target 1'** configuration dialog. Set the Xtal (Crystal frequency) frequency as 4 MHz (Fig. 1.12), and also the **Options for Target – Debug – use either Simulator / Keil Monitor- 51 driver** (Fig. 1.13). And also open the **Options for Target – Output** and click on the **'Create HEX File'** (Fig. 1.14).
9. If **Keil Monitor- 51 driver is used click on Settings** -> COM Port settings select the COM Port to which the board is connected and select the baud rate as 19200 or 9600

(recommended). Enable **Serial Interrupt** option if the user application is not using on- chip UART, to stop program execution.

10. Build the project; using **Project** -> **Build Target** (Fig. 1.15). µVision translates all the user application and links. Any errors in the code are indicated by – "Target not created" in the Build window, along with the error line. Debug the errors. After an error free, to build go to Debug mode.
11. Now user can enter into **Debug** mode with **Debug- Start / Stop Debug session** dialog. Or by clicking in the icon.
12. The program is run using the **Debug-Run** command & halted using **Debug-Stop Running**. Also the (reset, run, halt) icons can be used. Additional icons are

(step, step over, and step into, run till cursor).

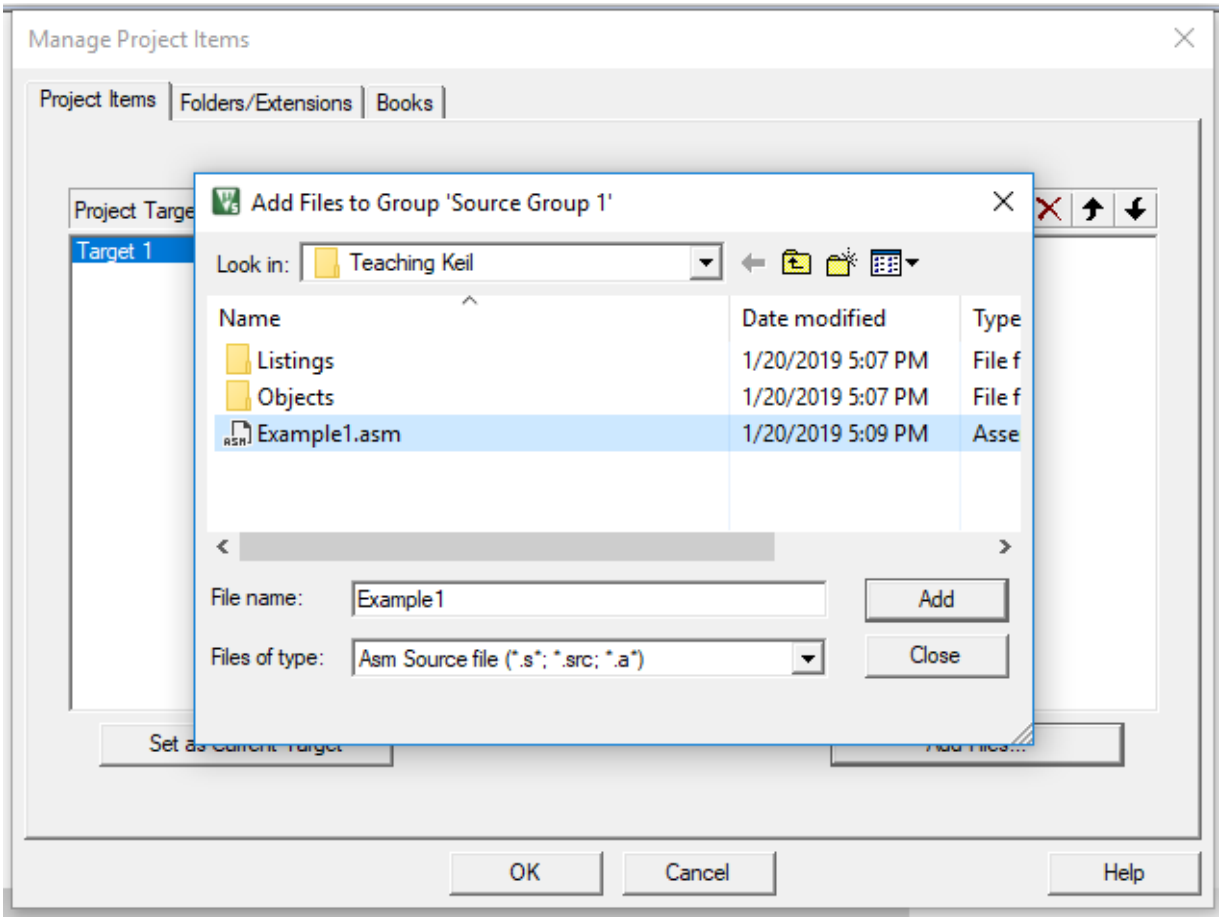


Figure 1.8

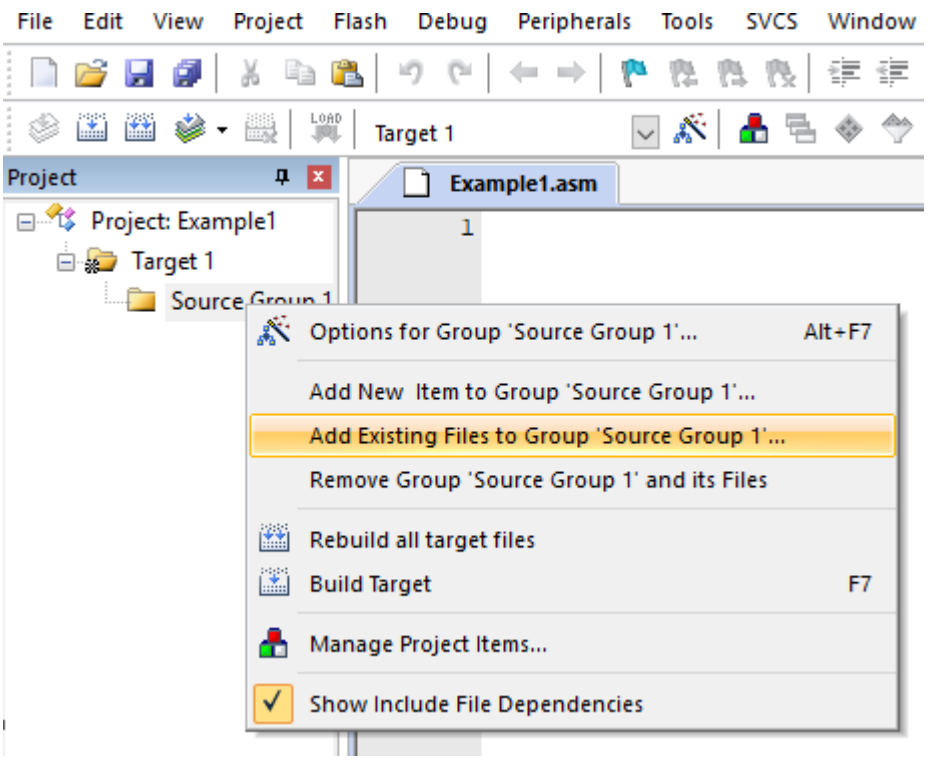


Figure 1.9

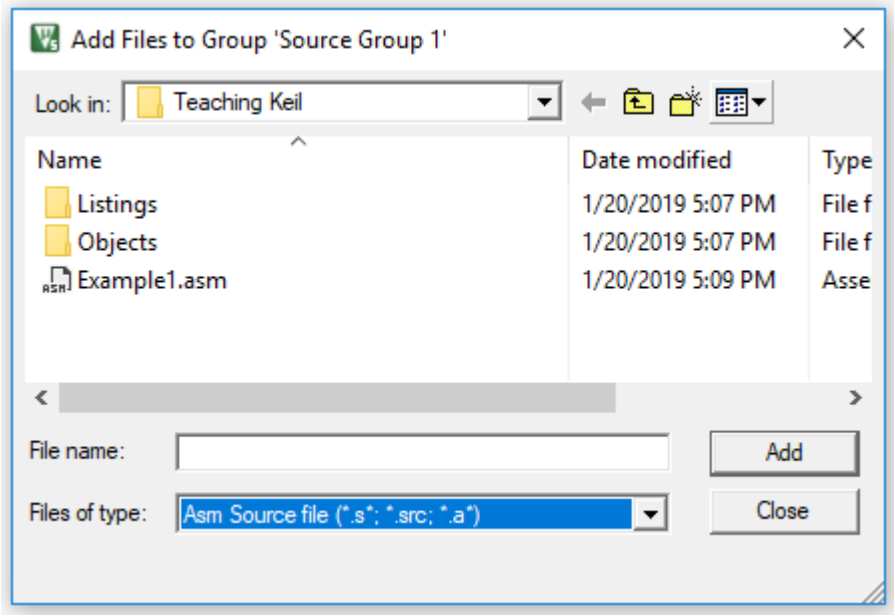


Figure 1.10

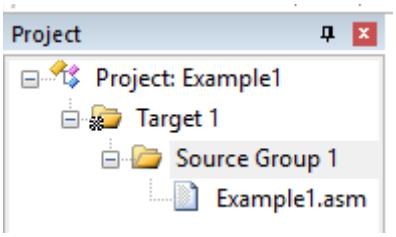


Figure 1.11 Program file Example1.asm added to Source Group 1

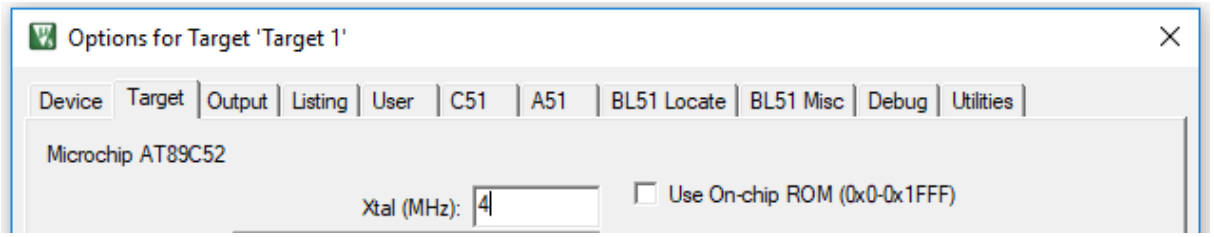


Figure 1.12

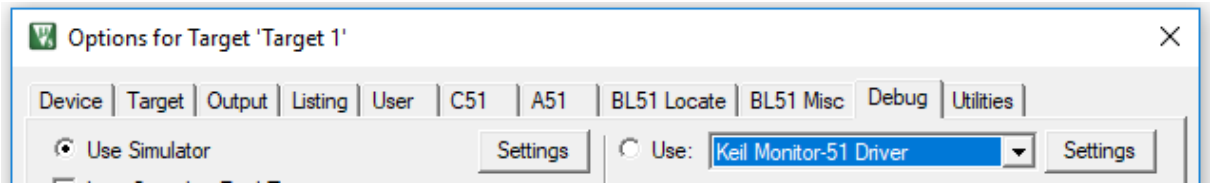


Figure 1.13

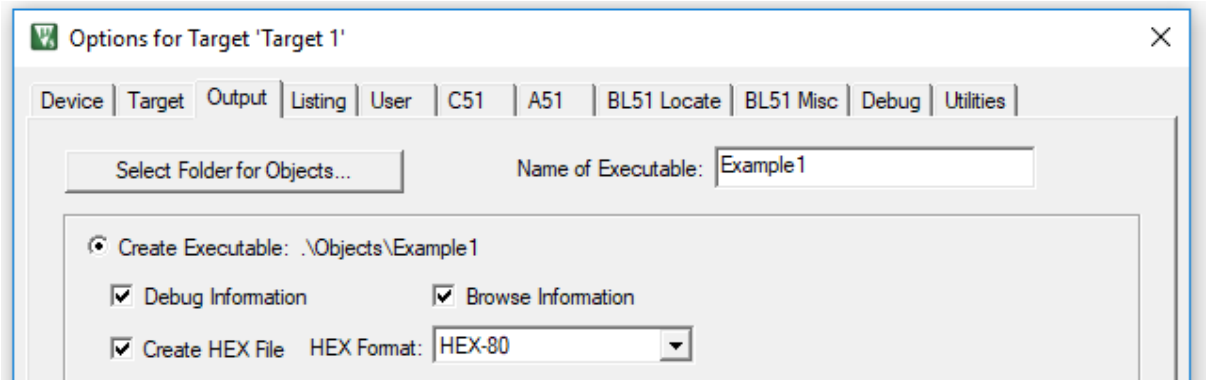


Figure 1.14

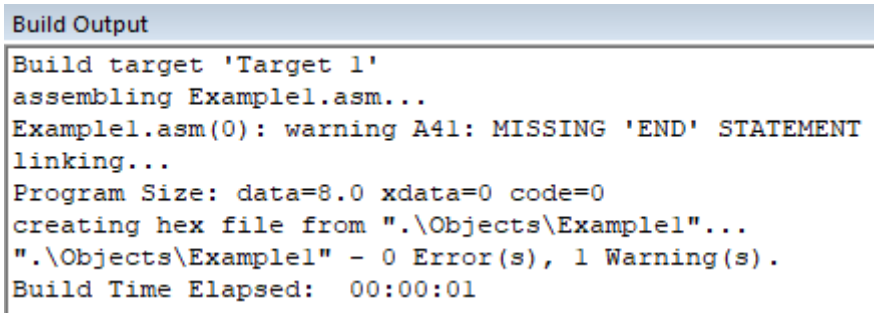


Figure 1.15

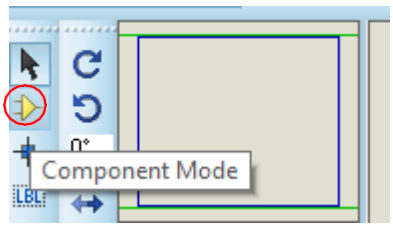
13. If it is an interface program the outputs can be seen on the LCD, CRO, motor, led status, etc. If it is a part-A program, the appropriate memory window is opened using View -> memory window (for data RAM & XRAM locations), Watch window (for timer program), serial window, etc.
14. Note: To access data RAM area type address as D: 0020h. Similarly to access the DPTR region (XRAM-present on chip in AT89C51ED2) say 9000h location type in X: 09000H.

The simulation of the project in Proteus

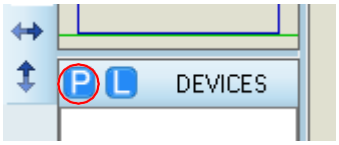
1. Double click on the Proteus 8 Professional icon on the desktop.



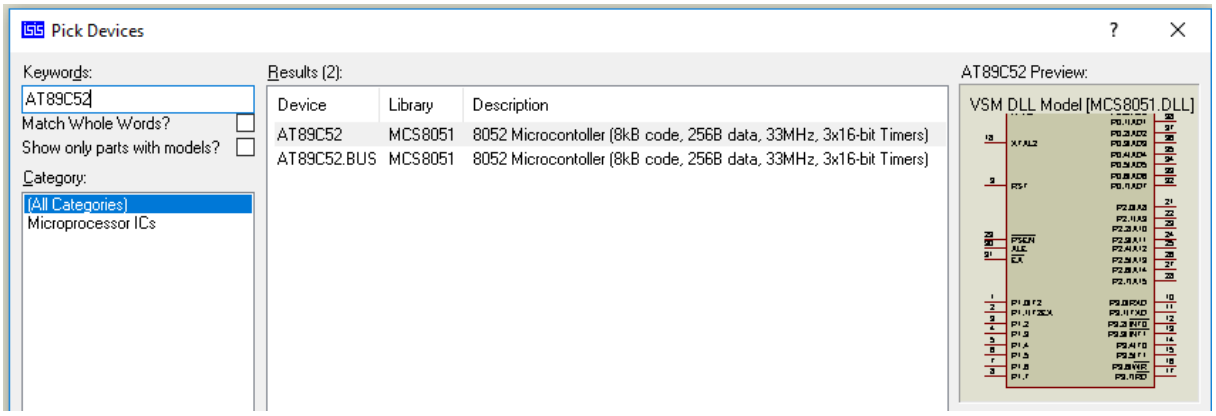
2. Click on “component mode” for component selection.



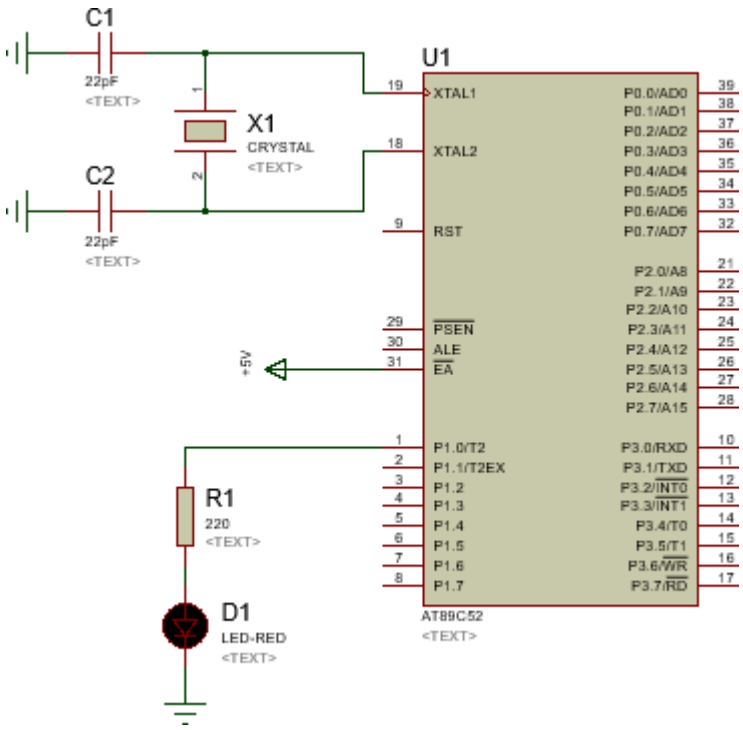
3. Click on “P”

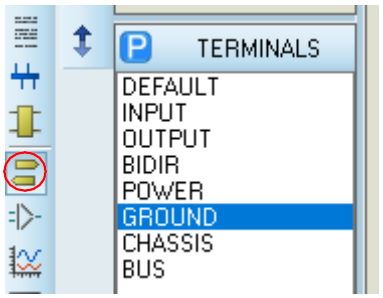


4. Pick device “AT89C52”. Type "CRYSTAL" to select the crystal, type "cap" to select the capacitor, type “res” to select the resistor and type “LED” to select the LED.

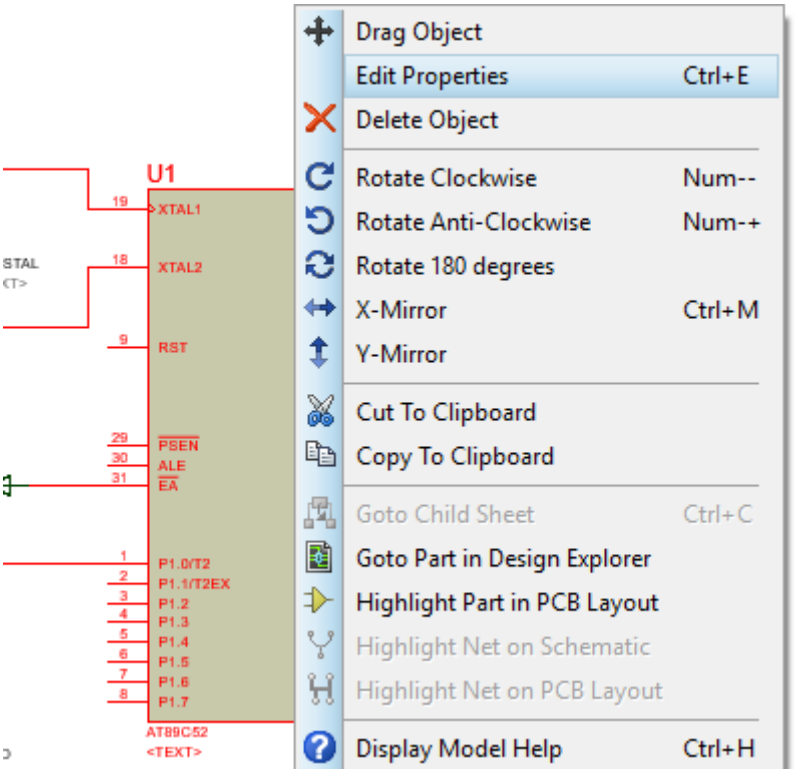


5. Connect the circuit as shown below. To add ground, click on “Terminal Mode” and choose “GROUND”.

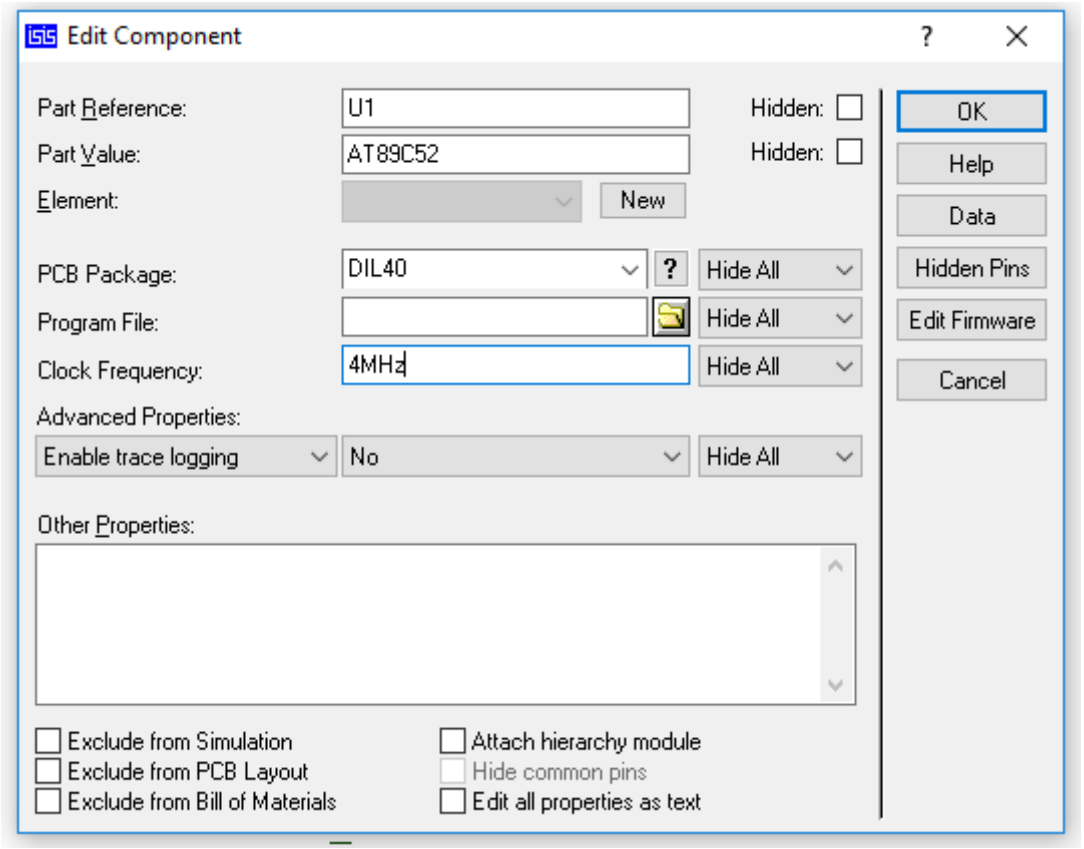




6. Right-click on AT89C52 and select "Edit Properties".



7. Edit clock frequency, choose HEX code from your computer and click on “OK”.



EXPERIMENT 1: Addressing Modes

Instructions tell the processor which operation to carry out. For example, MOV A, #5EH tells the processor to move the data 5EH to the accumulator.

Instructions result in the processor performing some operation on some data (the data being #5EH in the example above). The instruction above is an example of what's known as immediate addressing. The reason for this is because the data immediately follows the instruction in code memory.

However, it is not always adequate to store the data in code memory itself. If we needed to read the information from a keyboard, for example, we would need to use something other than immediate addressing.

There are eight different addressing modes in the 8051.

Opcode and Operand

The first byte of an instruction is known as the opcode (operation code) because this is the byte that is decoded by the processor - from this code the processor can work out what operation it must perform.

For a one-byte instruction there is only the opcode.

For a two-byte instruction the second byte is the operand. The operand can be data or an 8- bit address.

For a three-byte instruction the second and third bytes make up the operand. A two-byte operand is usually a 16-bit address, as we shall see when we look at long addressing.

Addressing Modes

The eight addressing modes are:

- Immediate
- Register
- Direct
- Indirect
- Relative
- Absolute
- Long
- Indexed

1.1. Immediate Addressing

If the operand is a constant then it can be stored in memory immediately after the opcode. Remember, values in code memory (ROM) do not change once the system has been programmed and is in use in the everyday world. Therefore, immediate addressing is only of use when the data to be read is a constant. For example, if your program needed to perform some calculations based on the number of weeks in the year, you could use immediate

addressing to load the number 52 (34H) into a register and then perform arithmetic operations upon this data.

```
MOV R0, #34H
```

The above instruction is an example of immediate addressing. It moves the data 34H into R0. The assembler must be able to tell the difference between an address and a piece of data. **The hash symbol (#) is used for this purpose** (whenever the assembler sees # before a number it knows this is immediate addressing).

This is a two-byte instruction.

If no H is appended to the number it is treated as decimal. For example, to place the decimal number 28 in R0, we write:

```
MOV R0, #28
```

If a hex number begins with a letter, it must be written with a leading zero. For example, to place A9H in R7, we write:

```
MOV R7, #0A9H
```

Exercise 1.1

Place the decimal number 47 into the R7 register. What would the contents of that register be after execution?

...

Exercise 1.2

Place the binary number 10010110 into the R1 register. What would the contents of that register be after execution?

...

Exercise 1.3

What are the hexadecimal and binary machine language bytes for the following instruction?

```
MOV R5, #84
```

...

With one exception, all instructions using immediate addressing use an 8-bit data constant for the immediate data. When initializing the data pointer, a 16-bit constant is required. For example,

```
MOV DPTR, #8000H
```

is a 3-byte instruction that loads the 16-bit constant 8000H into the data pointer.

1.2. Register Addressing

Often we need to move data from a register into the accumulator so that we can perform arithmetic operations upon it. For example, we may wish to move the contents of R6 into the accumulator.

```
MOV A, R6
```

This is an example of register addressing. It moves data from R6 (in the currently selected register bank) into the accumulator.

Note that the destination comes first. This is true of all instructions.

Exercise 1.4

Place the hexadecimal number E4H into the R5 register. After that move the contents of R5 into the accumulator.

...

...

1.3. Direct Addressing

Direct addressing is used for accessing data in the on-chip RAM. Since there are 256 bytes of RAM (128 bytes general storage for the programmer and another 128 bytes for the SFRs). That means the addresses go from 00H to FFH, any of which can be stored in an 8-bit location.

```
MOV A, 67H
```

The above instruction moves the data in location 67H into the accumulator. Note the difference between this and immediate addressing. Immediate addressing uses the data, which is immediately after the instruction. With direct addressing, the operand is an address. The data to be operated upon is stored in that address. **The assembler realises this is an address and not data because there is no hash symbol before it**.

```
ADD A, 06H
```

The above instruction adds the contents of location 06H to the accumulator and stores the result in the accumulator. If the selected register bank is bank 0 then this instruction is the same as ADD A, R6.

1.4. Indirect Addressing

Register addressing and direct addressing both restrict the programmer to manipulation of data in fixed addresses. The address the instruction reads from (MOV A, 30H) or writes to (MOV 30H, A) cannot be altered while the program is running.

There are times when it is necessary to read and write to a number of contiguous[1] memory locations. For example, if you had an array of 8-bit numbers stored in memory, starting at address 30H, you may wish to examine the contents of each number in the array (perhaps to find the smallest number). To do so, you would need to read location 30H, then 31H, then 32H and so on.

This can be achieved using indirect addressing. R0 and R1 may be used as pointer registers. We can use either one to store the current memory location and then use the indirect addressing instruction shown below.

```
MOV A, @Ri
```

where Ri is either R0 or R1.

Now, we can read the contents of location 30H through indirect addressing:

```
MOV R0,
#30H MOV
```

The first instruction is an example of immediate addressing whereby the data 30H is placed in R0. The second instruction is indirect addressing. It moves the contents of location 30H into the accumulator.

If we now wish to get the data in location 31H we use the following:

```
INC R0
MOV A, @R0
```

Once we see how to write a loop in assembly language, we will be able to read the entire contents of the array.

Indirect addressing is essential when stepping through sequential memory locations. For example, the following instruction sequence clears internal RAM from address 60H to 7FH:

```
        MOV R0, #60H
LOOP: MOV @R0,
#00H
        INC R0
        CJNE R0, #80H, LOOP
```

The first instruction initializes R0 with the starting address of the block of memory; the second instruction uses indirect addressing to move 00H to the location pointed at by R0 the third instruction increments the pointer (R0) to the next address; and the last instruction tests the pointer to see if the end of the block has been reached. The test uses 80H, rather than 7FH,

[1] Contiguous: Bitişik

because the increment occurs after the indirect move. This ensures the final location (7FH) is written to before terminating.

1.5. Relative Addressing

Relative addressing is used only with certain jump instructions. The system executes a jump by changing the contents of the PC to the address of the next instruction to be executed. For example, if we wished to jump to the instruction stored at location 4EH in code memory, the PC would be loaded with 4EH. Then, during the next execution cycle the contents of the PC (4EH) are placed on the address bus and the instruction at 4EH is retrieved.

A relative address (or offset) is an 8-bit signed value, which is added to the PC to form the address of the next instruction to be executed.

With 8-bit signed numbers, the MSB is used to determine whether the number is positive or negative. If the MSB is 0 then the number is positive, while if the MSB is 1 the number is negative.

The instruction below shows how to jump six locations ahead.

```
SJMP 06H
```

SJMP is an unconditional jump and is a 2-byte instruction. The number following it is an offset address. If this instruction were stored in code memory at locations 100H and 101H, as shown below:

100H 80H

101H 06H

The opcode for SJMP is 80H. The operand is the offset address. If this instruction were executed the PC would get the value 108H. This is what happens:

- The PC contains 100H, therefore the instruction 80H is read into the IR.
- The instruction is decoded as the 2-byte SJMP instruction.
- The PC is incremented so that the operand may be retrieved.
- The operand is read from code memory and the PC is incremented again (because this is a 2-bye instruction).
- The operand (06H) is added to the PC (102H + 06H = 108H).
- The next instruction (at 108H) is executed.

Once we deal with 2's compliment and how negative numbers are dealt with in the CPU, we will look at a backward jump.

The S in SJMP stands for short. The range of signed 8-bit numbers is -127 to 128. Therefore, using SJMP allows us to jump 127 locations forward or 128 locations backward. Hence the name short jump.

When writing assembly programs we do not need to calculate the offset when using SJMP. Instead, we use labels. If we wished to jump to the instruction at 108H we would simply label the instruction with an appropriate name, for example THERE. We would then write the assembly code SJMP THERE. The assembler does the work of replacing the label with the correct offset.

1.6. Absolute Addressing

Absolute addressing is only used with the ACALL and AJMP

instructions. ACALL - subroutine call (2 byte instruction)

AJMP - unconditional jump (2 byte instruction)

These instructions allow you to move to any instruction within the same 2K of memory.

We will look at the AJMP instruction only (at a later date, when we begin dealing with subroutines we will deal with the ACALL instruction). The operation of the AJMP instruction is detailed below:

AJMP

address (PC)

<- (PC) + 2

(PC_{10}-PC_0) <- $address_{10}$- $address_0$

Note that only the eleven least significant bits of the PC are altered. The five most significant bits remain the same. This means the AJMP will only allow you to jump to a location in the same 2K page as the instruction directly after the jump.

For example:

If the label THERE represents an instruction at address 0F46H and the instruction AJMP THERE is in memory at locations 0900H and 0901H, the assembler will encode the instruction as

111 00001 1st byte (A10 - A8 +

opcode) 01000110 2nd byte (A7 -

A0)

The underlined bits are the low-order 11 bits of the destination address, 0F46H = 00001111101000110B. The upper five bits in the program counter will not change when this instruction executes. Note that both the AJMP instruction and the destination are within the 2K page bounded by 0800H and 0FFFH, and therefore have the upper five address bits in common.

EXPERIMENT 2: Arithmetic and Logic Instructions

Arithmetic instructions operate on whole numbers only and support addition, subtraction, multiplication and division.

Addition

ADD A, #66H ; add the hex number 66 to the accumulator A

ADD A, 66H ; add to accumulator A the contents of RAM address 0066H

Since four addressing modes are possible, the ADD A instruction can be written in different ways:

ADD A, 7PH (direct addressing)

ADD A, @R0 (indirect addressing)

ADD A, R7 (register addressing)

ADD A, #35H (immediate addressing)

All arithmetic instructions execute one machine cycle except the INC DPTR instruction (two machine cycles) and the MUL AB and DIV AB instructions (four machine cycles). (Note that one machine cycle takes 1 μs if the 8051 is operating from a 12 MHz clock.)

Accumulator A is an SFR; it is an 8-bit register and its RAM address is 00E0H. A large number of instructions use accumulator A, but not all.

INC 66H ; increment (add 1) the contents of address 0066H

Example 2.1

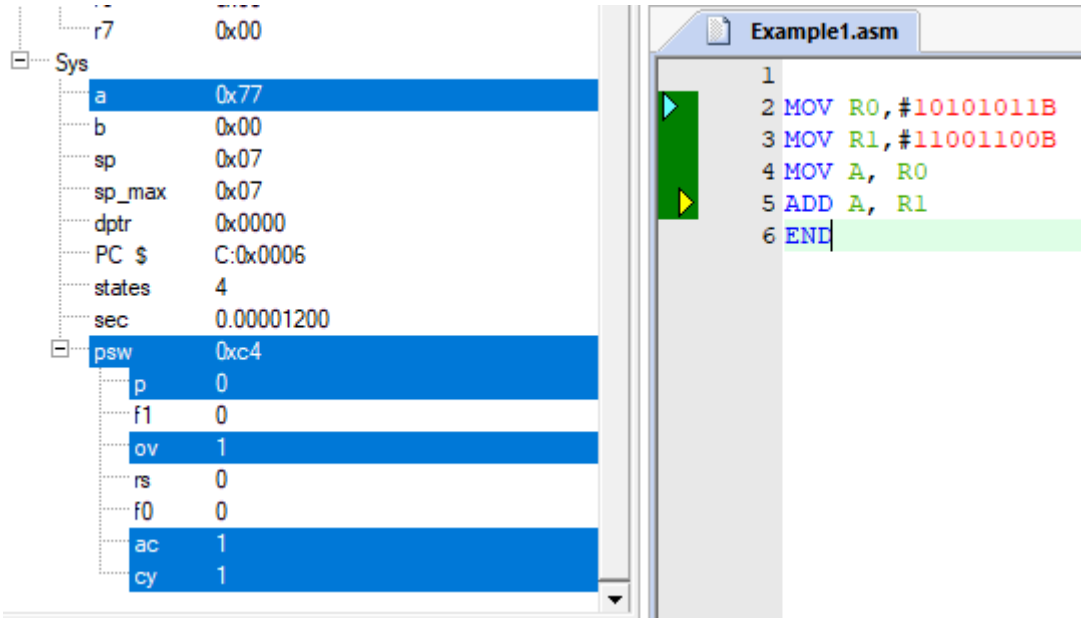
R0 contains 10101011 and R1 contains 11001100, and the PSW contains 00H. (a) What is the hexadecimal content of the accumulator and the PSW after execution of the following instruction?

```
MOV
R0,#10101011B
MOV
R1,#11001100B
MOV A, R0
```

(b) What is the content of the accumulator in decimal after execution of this instruction?

Solution

(a)



(b) 77H=7*16+7=119

Example 2.2

Suppose the 8051 did not have an instruction to directly increment an internal RAM location. How could this operation be achieved?

Solution

MOVE A,

direct INC A

MOV direct, A

Exercise 2.1

Is there any difference between the following two instructions?

A) INC A B) ADD A, #1

Subtraction

SUBB A, #66H ; subtract hex66 from the contents of A

The extra B in the instruction implies Borrow. If the contents of A are less than the number being subtracted then bit 7 of the program status word (PSW) SFR will be set.

DEC A ; decrement A by 1, put result into A

Exercise 2.2

Is there any difference between the following two instructions?

(1) DEC A (2) SUBB A,#1

Example 2.3

Illustrate an instruction sequence to subtract the content of R6 from R7 and leave the result in R7.

Solution

```
MOV A, R7
CLR C          ; clears the carry flag in the program status
word SUBB A, R6
MOV R7, A
```

Discussion

For both addition and subtraction, the accumulator holds one of the values for the operation. So, the first instruction above moves one of the bytes to the accumulator to prepare for the operation. The second instruction clears the carry flag in the program status word. This is required because the only form of the subtract instruction is SUBB—subtract with borrow.

The operation subtracts from the accumulator the source byte and the carry bit. For subtract operations, the carry bit serves as a "borrow" bit. If the state of the carry bit is unknown, it must be explicitly cleared using CLR C before executing SUBB. The third instruction performs the subtraction, leaving the result in the accumulator. The fourth instruction moves the result into R7.

The 8051 provides powerful addressing of its internal memory space. Any location can be incremented or decremented using direct addressing without going through the accumulator. For example, if internal RAM location 7FH contains 40H, then the instruction

INC 7FH

increments this value, leaving 41H in location 7FH.

Multiplication

MUL AB ; multiply the contents of A and B, put the answer in AB

A is the accumulator and B is another 8-bit SFR provided for use with the instructions multiply and divide. A and B are both 8-bit registers. The product of the multiplication process could be a 16-bit answer.

The MUL AB instruction multiplies the accumulator by the data in the B register and puts the 16-bit product into the concatenated[2] B (high-byte) and accumulator (low-byte) registers.

Example 2.4

A =135 decimal, B = 36 decimal. What would be the value in each register after executing the instruction MUL AB?

Solution

[2] Concatenated: Birleştirilmiş

A x B =4860 = 0001 0010 1111 1100B = 12FCH

0001 0010 or 12H would be placed in A; 1111 1100 or FCH in B

Example 2.5

The accumulator contains 55H, the B register contains 22H, and the program status word contains 00H. What are the contents of these registers after execution of the following instruction?

MUL A B

Solution

ACC=4AH, B=0BH, PSW=05H

Discussion

You might be able to compute the answer using a calculator; otherwise begin by expressing the original values in decimal: ACC = 55H = 85_{10} and B = 22H = 34_{10}. The product is 85_{10} x 34_{10}
= $2{,}890_{10}$ = 0B4AH. The high byte (0BH) is placed in register B and the low byte (4AH) is placed in the accumulator. The P bit in the PSW is set to establish even parity with the accumulator. Since the result is greater than 255_{10}, the overflow flag is set, leaving the PSW containing 05H.

Example 2.6

The accumulator contains 1FH. What is the largest value that could be in register B such that the OV bit would not be set after executing the following instruction?

MUL AB

Solution

08H

Discussion

The accumulator contains 1FH = 31_{10}. The overflow flag is set following MUL if the product is greater than 255_{10}. The largest number that 31_{10} can be multiplied by without the product exceeding 255_{10} is 8_{10} = 08H. (Note 31_{10} x 8_{10} = 248_{10}, 31_{10} x 9_{10} = 279_{10}.)

Exercise 2.3

If A = 2FH and B = 02H, what would each register contain after execution of the instruction MUL AB?

Division

DIV AB ; divide A by B, put quotient in A and remainder in B

Example 2.7

A = 135, B = 36. What would be the value in each register after execution of the instruction DIV AB?

Solution

Decimal values are assumed if the value quoted is not followed

by an H A/B =3; remainder 27 (27 = 1BH) : Hence 03H in A, 1BH

in B

Exercise 2.4

If A = 2FH and B = 02H, what would each register contain after the execution of the instruction DIV AB?

LOGIC OPERATIONS

The set of logic functions include:

ANL AND

Logic ORL

OR Logic

XRL exclusive OR Logic

CPL Complement (i.e. switch to the opposite logic

level) RL Rotate Left (i.e. shift byte left)

RR Rotate Right (i.e. shift byte

right) SETB Set bit to logic 1

CLR Clear bit to logic 0

AND Operation

The ANL instruction is useful in forcing a particular bit in a register to logic 0 whilst not altering other bits. The technique is called masking.

Suppose accumulator contains EDH (1110 1101B),

1	1	1	0	1	1	0	1

bit 1 and bit 4 are at logic 0, the rest at logic 1.

ANL R1, #7FH ; 7FH = 0111 1111B, forces bit 7 to zero

1	1	1	0	1	1	0	1

AND

0	1	1	1	1	1	1	1

=

0	1	1	0	1	1	0	1

Exercise 2.5

If A = 2D, what would be the accumulator contents after execution of the instruction ANL A, #3BH?

OR Operation

Another aspect of masking is to use the ORL instruction to force a particular bit to logic 1, whilst not altering other bits.

Suppose accumulator contains A2H (1100 0010B),

1	0	1	0	0	0	1	0

bit 0, 2, 3, 4 and bit 5 are at logic 0, the rest at logic 1.

ANL A, #14H ; 14H = 0001 0100B, forces bit 2 and 4 to one

1	0	1	0	0	0	1	0

OR

0	0	0	1	0	1	0	0

=

1	0	1	1	0	1	1	0

A=B6H

The power control (PCON) SFR in the 8051 family, is not bit addressable and yet has a couple of bits that can send the microcontroller into idle mode[3] or power down mode, useful when the power source is a battery.

The contents of the PCON SFR are:

SMOD1	SMOD2		POF	GPF1	GPF2	PD	IDL

SMOD1 and 2 are used when setting the baud rate of the serial onboard peripheral. POF, GPF1 and GPF2 are indictor flag bits. IDL is the idle bit; when set to 1 the microcontroller core goes to sleep and becomes inactive. The on-chip RAM and SFRs retain their values. PD is the Power Down bit, which also retains the on-chip RAM and SFR values but saves the most power by stopping the oscillator clock.

ORL PCON,#02H ; enables Power

Down ORL PCON,#01H ; enables

Idle mode

Either mode can be terminated by an external interrupt signal.

Exercise 2.6

[3] Idle mode: Bekleme modu

If the contents of accumulator = 38H, what would the contents of that register be after execution of the following instruction?

ORL A, #9AH

CPL Complement Operation

The instructions described so far have operated on bytes (8 bits) but some instructions operate on bits and CPL is an example.

CPL P1^7 ; complement bit 7 on Port 1

Port 1 is one of the microcontroller's ports with 8

pins. Port 1

P1^7	P1^6	P1^5	P1^4	P1^3	P1^2	P1^1	P1^0
MSB						LSB	

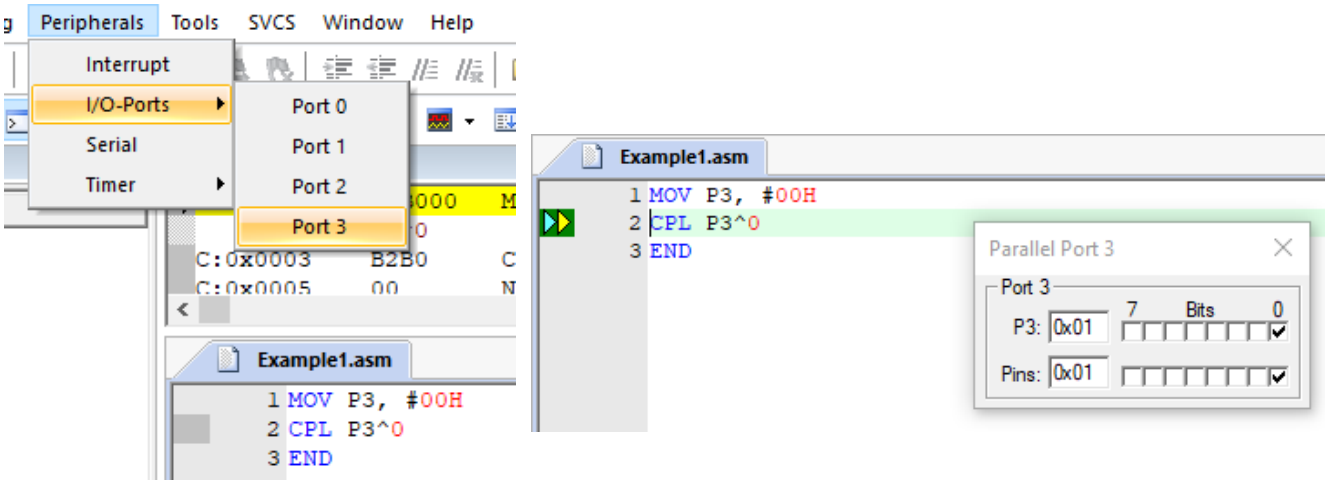
Example 2.8

What would be the value in port3 after executing the following instruction?

```
MOV P3,
#00H CPL
P3^0
```

Solution

To open the Port screen, click on **Peripherals** -> **I/O-Ports** -> **Port 3**



P1=01H

Exercise 2.7

Is there any difference between the following two instructions? (MOV P2, #00H)

1) CLR P2^5 2) CPL P2^5

RL, rotate left one bit, RR, rotate right one bit operations

Suppose the accumulator A contents are 0000 0001B; this is 01H. RL A ; contents of A become 0000 0010B or 02H RL A ; 0000 0100B or 04H RL A ; 0000 1000B or 08H

RL three times has the effect of multiplying A by 2^3 i.e. by 8.

Suppose the accumulator A contents are 1000 0000B, or 128 decimal, then: RR A ; contents of A become 0100 0000B which is 64 decimal RR A ; A becomes 0010 0000B=32 decimal RR A ; A becomes 0001 0000B=16 decimal RR A ; A becomes 0000 1000B=8 decimal

RR four times has the same effect as dividing A by 2^4

i.e. 16. 128/16=8

Exercise 2.8

If the content of A is 128 and B is 2, what would the register contents be after execution of the following instructions?

```
RR
A
RL
B
RR
```

SETB set bit, CLR clear bit operations

This instruction operates on a bit, setting it to logic 1.

```
SETB P1^7    ; set bit 7 on Port 1 to
logic 1 CLR P1^7    ; clears bit 7 on port
```

EXPERIMENT 3: Data Transfer, Branch Operations and Time Delays

MOV Operation

MOV moves bytes of data. Consider driving a seven-segment display (decimal point dp included) where each LED is driven by the sink method. See Figure 3.1.

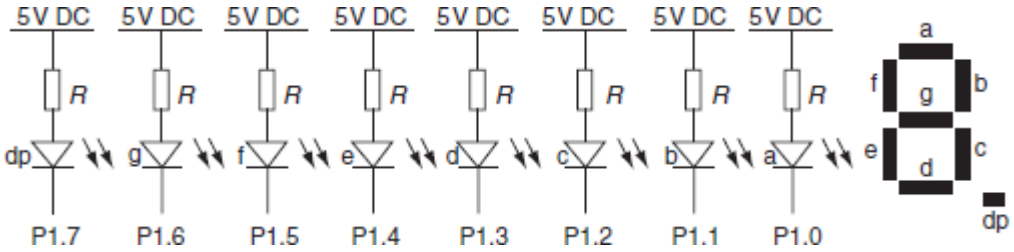


Figure 3.1

Each LED illuminates a segment. The seven-segment display is shown to the right with its standard segment identification letters.

Example 3.1 Write two program lines, one to display 5, the second to display 2. In both cases turn the decimal point off.

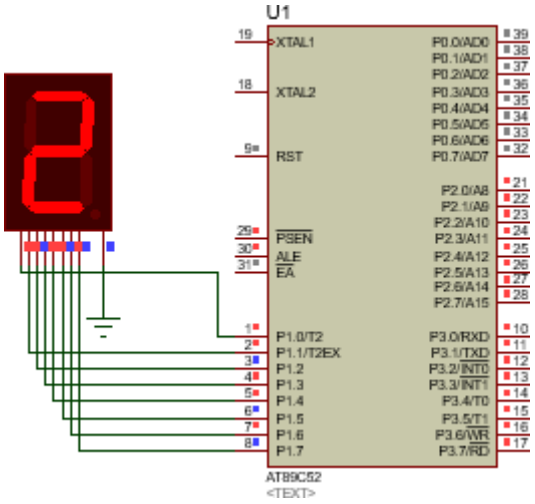
Solution

	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	
	dp	g	f	e	d	c	b	a	
5	0	1	1	0	1	1	0	1	6DH
2	0	1	0	1	1	0	1	1	5BH

MOV P1,

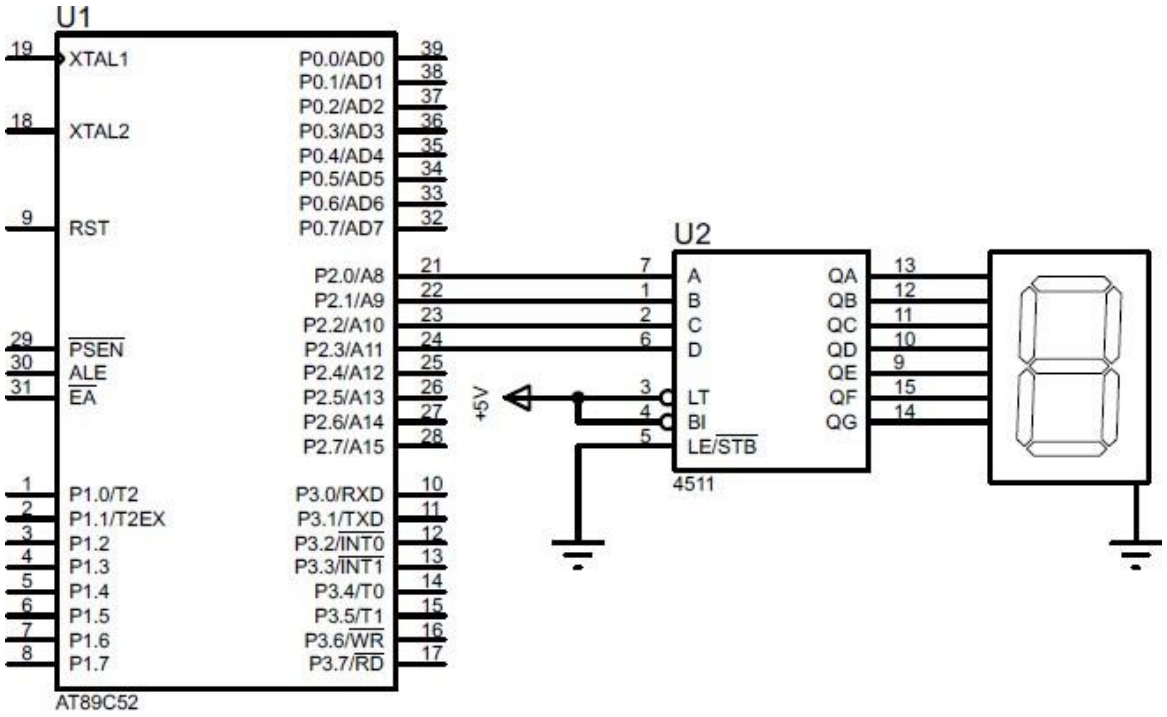
#6DH MOV

P1, #5BH



Exercise 3.1 Write two program lines, one to display 7, the second to display 4. In both cases turn the decimal point off.

Example 3.2 Consider the following circuit and write the code that shows the number 5 on the display.



Solution

```
        ORG 00H
        MOV P2,
        #5D END
```

Conditional Jumps

JC <address>

If the carry is equal to 1, branch to the indicated address.

Example 3.3 The accumulator contains F1H, the B register contains 42H. Write a program code that sum this two numbers. And if carry is equal to 1, move the A1H in R1 register. If not, move the B5H in R7 register.

Solution

```
        MOV A,
        #0F1H
        MOV B,
        #42H ADD
        A, B                ; If the carry is equal to 1, branch to the
        JC here
        MOV R7,
#0B5H here: MOV
```

Exercise 3.2 The R0 register contains AEH, the R7 register contains 85H. Write a program code that sum this two numbers. And if carry is equal to 1, move the 0FH in P0 port. If not, move the F0H in P2 port.

JB jump ; jump if bit = 1

JNB jump ; jump if bit = 0

Consider a practical example of testing switched logic levels. Refer to Figure 3.2. If the switch is not pressed the voltage on the port pin is 0 V. When the switch is pressed and held, then the port pin is connected directly to 5 V. To test for switch being pressed, the following program could be used:

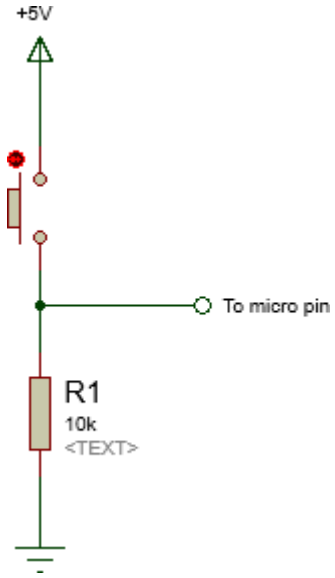
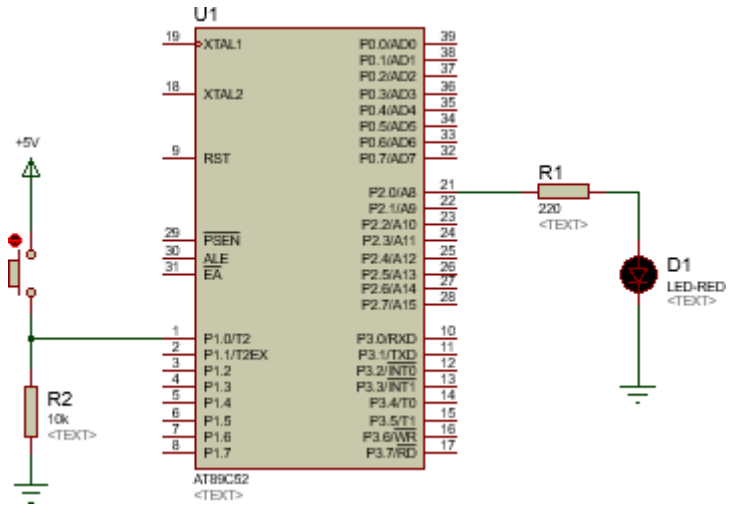


Figure 3.2 Circuit to produce logic levels 0 or 1 at a port pin. Circuit normally producing logic 0

```
        ORG 0                   ; sets start address to 0
STAR    JB P1^0,                ; jump to DoSomething if pin 0 port 1 is
        DoSomething SJMP        logic 1
DoSomething:
```

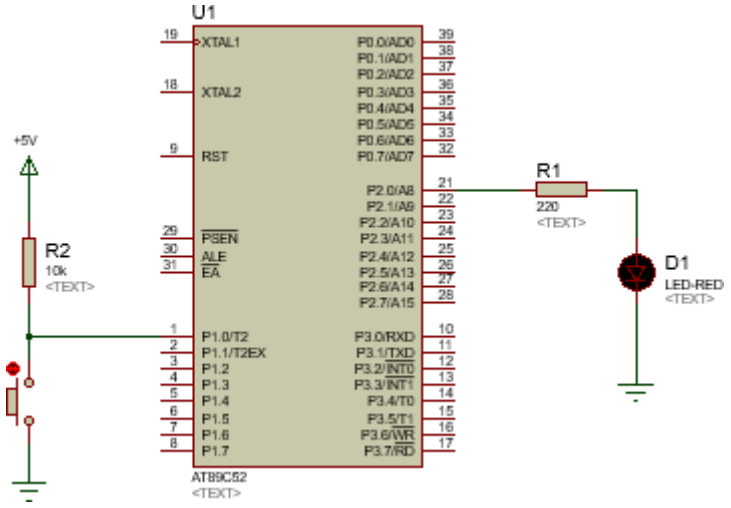
Example 3.4 A button connected to the pin 0 of port 1 and a LED connected to the pin 0 of port 2. Write a program code that if the switch is pressed, then turn on the LED.

Solution




```
                ORG 0                   ; sets start address to 0
                MOV P1,                 ; reset port 1
                #00H MOV                ; reset port 2
STAR            P2, #00H
                JB P1^0,                ; jump to DoSomething if pin 0 port 1 is
                                        logic 1
DoSomething: SETB                       ; go to START check switch
P2^0
```

Exercise 3.3 A button connected to the pin 0 of port 1 and a LED connected to the pin 0 of port 2 (following figure). Write a program code that if the switch is pressed, then turn on the LED.



JNC <address>

If the carry is equal to 0, branch to the indicated address.

Example 3.5 The R4 register contains 28H, the B register contains 17H. Write a program code that multiply this two numbers. And if carry is equal to 0, move the 04H in P1 port. If not, move the 47H in P2 port.

Solution

```
        MOV R4,
        #28H MOV
        B, #17H
        MOV A, R4
        MUL AB
        JNC jump     ; If the carry is equal to 0, branch to the ‘here’.
        MOV P2,
#47H jump: MOV
R1, #04H
```

JZ <address> ; Jump if the accumulator is equal to 0

This instruction tests the contents of the accumulator. If the contents are equal to 0 it causes a jump to the indicated address, otherwise execution continues with the next instruction.

JNZ <address> ; Jump if the accumulator is not equal to 0

This instruction also tests the contents of the accumulator. However, if the contents are NOT equal to 0 it causes a jump to the indicated address, otherwise execution continues with the next instruction.

Example 3.6 The accumulator contains 08H. Decrease the accumulator until it become zero.

Solution

```
        MOV A, #08H
jump: SUBB A, #01H
        JNZ
        jump
```

Exercise 3.4 The accumulator contains 10H. Decrease the accumulator until it become zero. Use DEC command.

DJNZ <address> ; Decrement and jump if not zero

Example 3.7 The R1 register contains 0FH. Decrease the R1 register until it become zero.

Solution

```
    MOV R1, #0FH
jump:  DJNZ R1,
              jump
```

The $ Operator

Most assemblers use the $ character as a pointer to the current program location. Therefore, the instruction DJNZ R0, $ means - decrement the contents of R0 and if the result is non-zero

jump back to this line. In this way, the contents of R0 are continuously decremented until they reach zero.

Example 3.8 The R7 register contains 04H. Decrease the R7 register until it become zero.

Solution

```
MOV R7,
#04H DJNZ
R7, $ END
```

Unconditional branch operations

are ACALL absolute call

LCALL long call

ACALL calls up a subroutine, the subroutine must always have RET as its last operation. ACALL range is limited to +127 places forward or -128 places backward. If your program jumps further than ACALL the compiler will report that the program is jumping out of bounds and replacement by LCALL will solve the problem.

ACALL is two bytes long, LCALL is three bytes

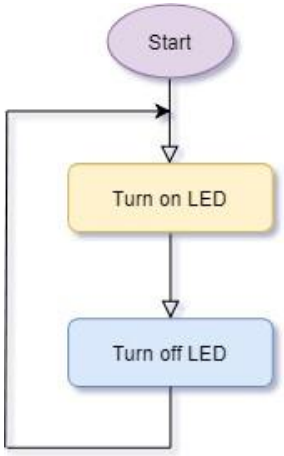
long. Consider an example (a subroutine called

by ACALL):

```
DELAY:    MOV R0,#34      ; move decimal 34 into register R0
TAKE:     DJNZ R0,TAKE    ; keep subtracting 1 from R0 until zero
          RET             ; return from subroutine
```

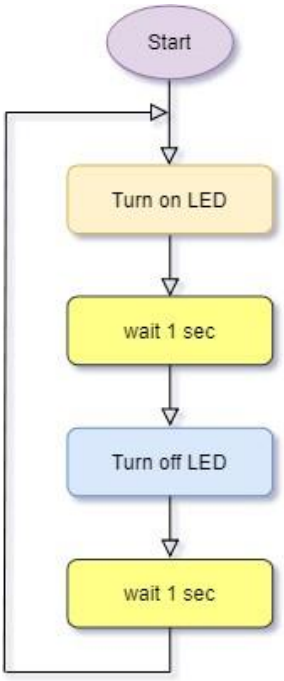
Why We Need Time Delays?

There are times when it is necessary to wait before executing the next part of a program. For this discussion we will take the example of flashing an LED.



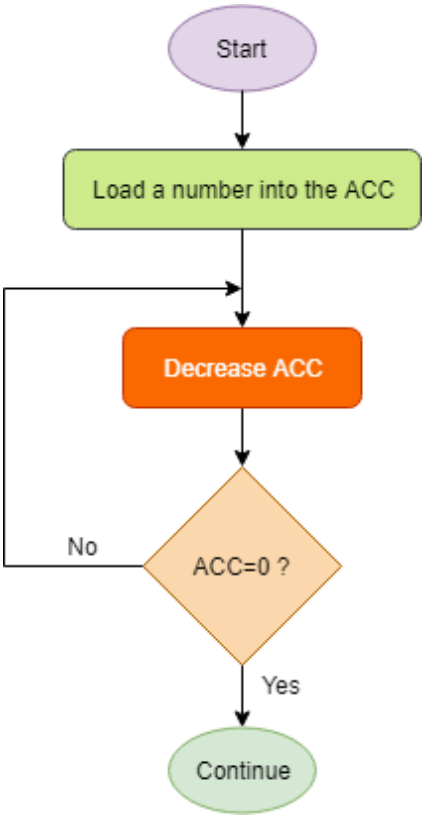
The above flowchart details program for flashing an LED. It works by turning on the LED, turning it off and then looping back to the start again. This program is in a never-ending loop.

However, while the above program may flash an LED, because of the speed at which microcontrollers operate, the frequency would be too fast for the human eye to see.



Above is a modification of the LED flashing program. It turns on the LED, waits a second, turns it off, waits another second and then loops back to the start again.

One method of forcing the microcontroller to wait is the time delay. The flow chart for a single- loop time delay is shown below.



A number is loaded into the accumulator. It is then decremented. Following this, a check on the contents of the accumulator is made. If the contents of the accumulator is not equal to zero a jump back to the decrement instruction is made.

If 0AH is loaded into the accumulator, then the loop is executed ten times. To increase the length of the time delay we load a larger number into the accumulator. Therefore, the longest time delay (using this single loop time delay) is achieved by loading the accumulator with FFH
- the loop would be executed 255 times.

Now there is a question to be answered. One execution time is how many seconds?

Knowledge of the microcontroller clock cycle time is useful in defining timing events used in applications.

Some AT89 Instruction Set and Their Oscillator Periods

Mnemonic	**Oscillator Period**
MOV	12
ADD	12
SUBB	12
INC	12
DEC	12
MUL	48
DIV	48
ANL	12
ORL	12
XRL	12
JC	24
JNC	24
JB	24
JNB	24
JBC	24
ACALL	24
LCALL	24
RET	24
RETI	24
JMP	24
JZ	24
JNZ	24
DJNZ	24
NOP	12
CLR	12
SETB	12

Example 3.9 A AT89C52 microcontroller has a clock frequency of 11.0592 MHz. What is the time for each cycle?

Solution

$$\text{Clock period } (T) = \frac{1}{\text{Frequency}} = \frac{1}{11.0592 \times 10^6} = 90.4225 \text{ ns } (1 \text{ ns} = 10^{-9} \text{ s})$$

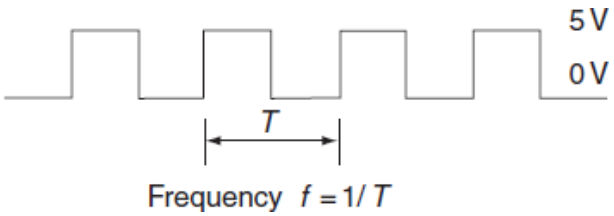


Figure 3.3 Square-wave signal at a frequency f Hz

Time Delays

The NOP instruction is a simple time delay but apart from this there are two methods of creating time delays:

- register decrement
- onboard timers

The use of onboard timers will be described in a later experiment; here the register decrement method will be described.

The basic single loop program lines are:

```
    MOV R0, #number ; move a number into an 8-bit register
    R0 DJNZ R0, $ ; keep decrementing R0 until it is zero
```

MOV takes 12 clock cycles, DJNZ takes 24 clock cycles. The delay time is (12+(numberx24))xClock cycles.

Delay time = (12+(numberx24))xClock cycles

Example 3.10 A AT89C52 microcontroller has an 11.0592 MHz crystal-controlled clock oscillator. Write an assembly language program that will generate a 5 kHz squarewave signal on pin 7 of port 1.

Solution

Clock frequency = 11.0592 MHz

Thus period of clock cycle = (1/11.0592 MHz)

=90.4225 ns Signal frequency = 5 kHz

Therefore period of signal cycle = (1/5 kHz) = 200 μs

The delay required is half of this value since the square wave has an equal logic 1/logic 0 time. See Figure 3.3.

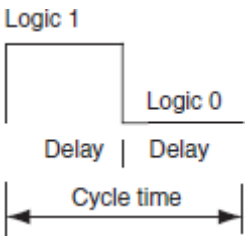


Figure 3.3 Delay period determination for a square-wave

signal Delay = 100 μs = (12+(numberx24))x90.4225ns

Hence, number = ((100 μs/90.4225ns)-12)/24=45.58 ≅ 46 decimal (to the nearest whole number).

```
        ORG 0              ; sets start address to 0
jump:   SETB P1^7          ; set pin 7 on port 1 to logic 1
        MOV R0,            ; move 46 decimal number into an 8-bit
        #46D DJNZ          register R0
        R0, $ CLR          ; keep decrementing R0 until it is zero
        P1^7 MOV           ; clear pin 7 on port 1 to logic 0
        R0, #46D           ; move 46 decimal number into an 8-bit
        DJNZ R0, $         register R0
        JMP                ; keep decrementing R0 until it is zero
        jump               ; it takes 24 clock cycles
```

Exercise 3.5 Using the techniques above, assuming the clock frequency is 11.0592 MHz, write a program to generate a pulse of 20 kHz on pin 5 of port 2 of the microcontroller.

Delay with using subroutine

```
DELAY:    MOV R0,#34       ; move decimal 34 into register R0
TAKE:     DJNZ R0,TAKE     ; keep subtracting 1 from R0 until zero
          RET              ; return from subroutine
```

MOV takes 12 clock cycles, DJNZ and RET each take 24 clock cycles. The delay is called up from the main program using ACALL, which takes 24 clock cycles. The delay time is $(24_{ACALL}+12_{MOV}+(numberx24)_{DJNZ}+24_{RET})$xClock cycles.

Delay time = (24+12+(numberx24)+24)xClock

cycles Delay time = (60+(numberx24))xClock

cycles

Example 3.11 A AT89C52 microcontroller has an 11.0592 MHz crystal-controlled clock oscillator. Write an assembly language program that will generate a 8 kHz squarewave signal on pin 3 of port 1.

Solution

Clock frequency = 11.0592 MHz

Thus period of clock cycle = (1/11.0592 MHz)

=90.4225 ns Signal frequency = 8 kHz

Therefore period of signal cycle = (1/8 kHz) =

125 μs Delay = 125μs/2 =

(60+(numberx24))x90.4225ns

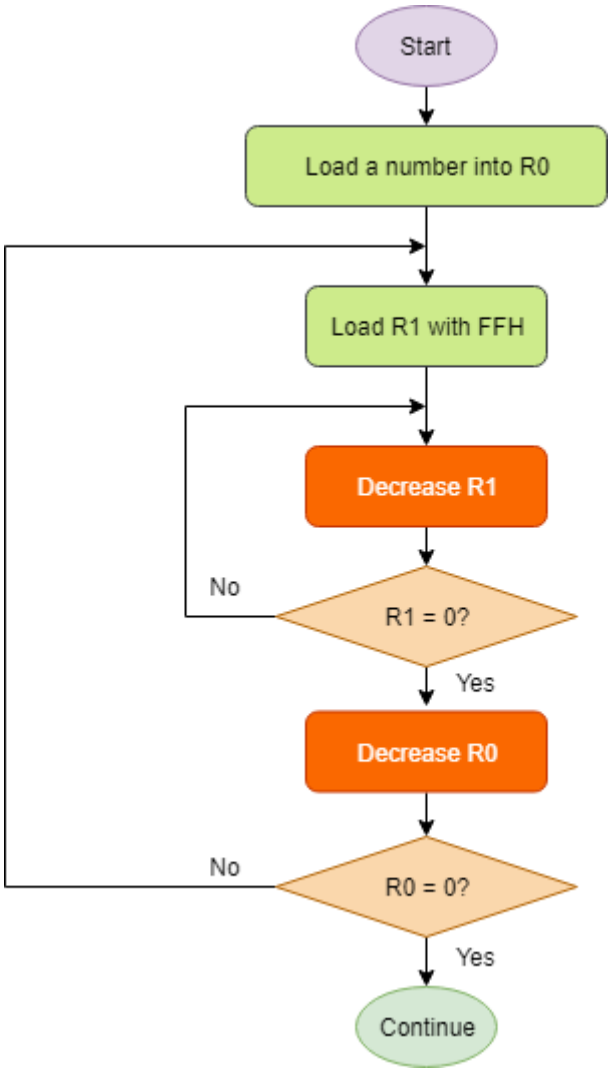
Hence number = ((125μs/(2x90.4225ns))-60)/24=26.3≅ 26 decimal (to the nearest whole number).

```
        ORG 0           ; sets start address to 0
START:  SETB P1^3       ; set pin 3 on port 1 to logic 1
        ACALL DELAY
        CLR P1^3        ; clear pin 3 on port 1 to logic 0
        ACALL DELAY
        JMP START       ; go to START
DELAY:  MOV R0, #26D    ; move a number into an 8-bit register R0
TAKE:   DJNZ R0, TAKE   ; keep decrementing R0 until it is zero
        RET             ; return from DELAY subroutine
        END             ; no more assembly language
```

Exercise 3.6 Using the techniques above, assuming the clock frequency is 11.0592 MHz, write a program to generate a pulse of 2.5 kHz on pin 2 of port 1 of the microcontroller.

Longer Delays

The longest single-loop delay is achieved by initialising the decrement register (ie; the register that will be decremented) to FFH. To achieve longer delays we can use multiple-loop time delays, as detailed in the flow chart below.



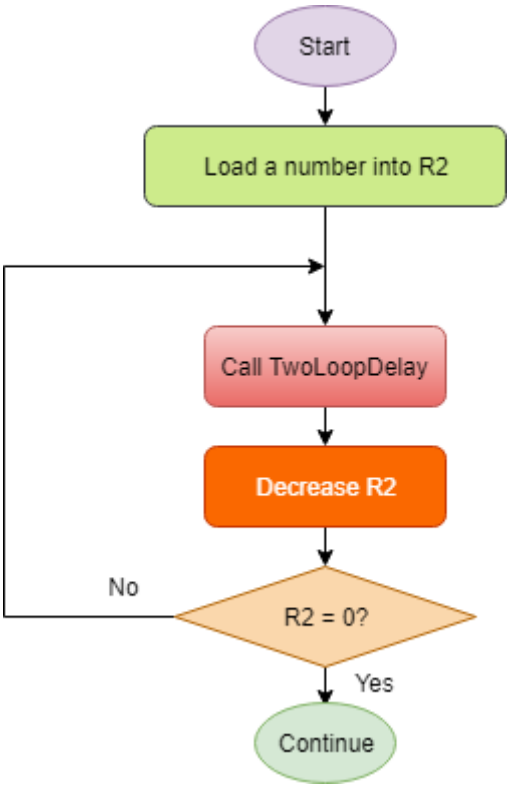

```
TwoLoopDelay:   MOV R0, #number  ;Load a number into R0
InnerLoop:      MOV R1, #0FFH    ; move FFH into an 8-bit register R1
                DJNZ R1, $       ; keep decrementing R1 until it is zero
                DJNZ R0,
                InnerLoop
                RET              ; return from TwoLoopDelay subroutine
                END              ; no more assembly language
```

The inner loop (nested loop) takes 255 iterations before R1 reaches 0. When this happens, the loop is exited and R0 is decremented. If R0 is not equal to zero the program jumps back to the line that loads R1 with FFH again. Another 255 iterations of the nested loop must occur before R0 is decremented again. Therefore, for every one iteration of the outer loop there are 255 iterations of the inner loop.

We control the length of the time delay by changing the value loaded into R0. If R0 is initialised to 03H, then the overall number of iterations is 3 * 255 = 765. The maximum number of iteration for a two-loop time delay is 255 * 255 = 65025.

With 11.0592 MHz crystal-controlled clock oscillator, we can obtain maximum 65025x90.4225ns=5.8797 ms delay.

If we want the more delay we should use three loop.




```
DELAY:      MOV R2, #number  ;Load a number into
CallAgai    ACALL
            TwoLoopDelay
            DJNZ R2,
            CallAgain RET
TwoLoopDela MOV      R0,     ; move FFH into an 8-bit register
y:          #0FFH   MOV      R0
            R1,     #0FFH    ; move FFH into an 8-bit register
            DJNZ R1, $       R1
            DJNZ R0,
            InnerLoop RET    ; return from TwoLoopDelay
                             subroutine
```

Example 3.12 A AT89C52 microcontroller has an 11.0592 MHz crystal-controlled clock oscillator. A LED connected to the pin 0 of port 2. Write an assembly language program that will turn on the LED 0.5 s and turn off the LED 0.5s.

Solution: We need 0.5 s delay,

Delay = 0.5s =

(60+(numberx24))x90.4225ns

number=230400

240x160x60=230400

Program:

```
                ORG 0                ; sets start address to 0
```

```
START:      SETB P2^0          ; set pin 0 on port 2 to
            ACALL
            DELAY
            CLR P2^0           ; clear pin 0 on port 2 to
            logic 0
            ACALL
DELAY:      DELAY              ; go to START
CallAgai    JMP
            START
            MOV R2,
            #6D
TwoLoopDela ACALL              ; move 160D into an 8-bit register
y:          TwoLoopDelay       R0
            DJNZ R2,           ; move 240D into an 8-bit register
            CallAgain RET
            MOV        R0,     ; return from TwoLoopDelay
            #160D   MOV        subroutine
```

EXPERIMENT 4: The 8051 Timers

The basic 8051 has two on-chip timers that can be used for timing durations or for counting external events (AT89C52 has three on-chip timers).

Interval timing allows the programmer to perform operations at specific instants in time. For example, in our LED flashing program the LED was turned on for a specific length of time and then turned off for a specific length of time. We achieved this through the use of time delays. Since the microcontroller operates at a specific frequency, we could work out exactly how many iterations of the time delay was needed to give us the desired delay.

However, this is cumbersome and prone to error. And there is another disadvantage; the CPU is occupied, stepping through the loops. If we use the on-chip timers, the CPU could be off doing something more useful while the timers take on the menial task of keeping track of time.

Timers 0 and 1

Timers 0 and 1 are fundamentally the same and both have two 8-bit registers, timer high (TH) byte and timer low (TL) byte. Both share the timer control (TCON) register and the timer mode (TMOD) register.

The timers can be configured into one of the four modes:

- **Mode 0** TH and TL come together to form a 13-bit register where TH has 5 bits. This makes the microcontroller compatible with an earlier device.
- **Mode 1** TH and TL come together to form a 16-bit register.
- **Mode 2** TL is the working 8-bit register and TH is the automatic reload register. This mode is used to define the baud rate of the serial UART interface.
- **Mode 3** TH and TL registers of both timers combine to produce three 8-bit

timers. Details of the TMOD SFR are:

TMOD

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

This register has two identical halves, the upper four bits for timer 1 and the lower four bits for timer 0. The bits M1 and M0 set the TMOD:

M1	M0	
0	0	Mode 0
0	1	Mode 1
1	0	Mode 2
1	1	Mode 3

MODE 1 16-BIT UP COUNTER

Figure 4.1 shows the timer in mode 1 with the TH and TL registers together forming a

16-bit register. The diagram also shows that for this microcontroller family, **the timer clock is one-**

twelve of the microcontroller clock. The default value of TMOD is 00H, so C/T=0 and the peripheral defaults to being a timer rather than a pulse counter. The default value of GATE is 0 and this is inverted, so the OR output defaults to logic 1. The timer is turned on or off by TR1/0 (in TCON register), putting TR0=1 would turn timer 0 on.

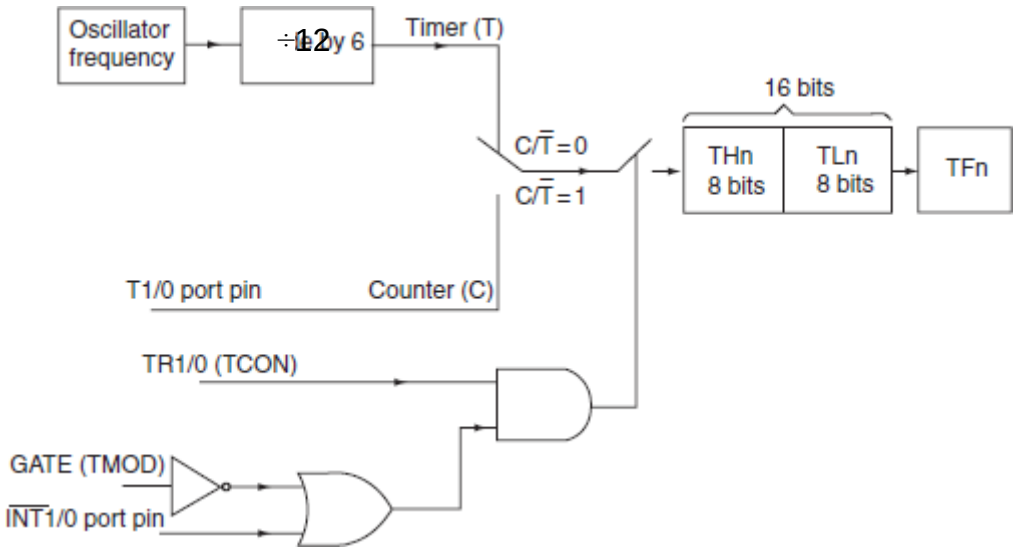


Figure 4.1 Circuit for timer 0/1 operating as a 16-bit up-counter in mode 1

In mode 1, the TH and TL registers in timer 0 or timer 1 join to form a 16-bit up counter. The counter can be loaded with a base number from which the timer can increment upwards towards the 16-bit maximum of 65535 (FFFFH). The time taken to count from the base number to the maximum count value is the required delay. Figure 4.2 shows the method of achieving the required delay.

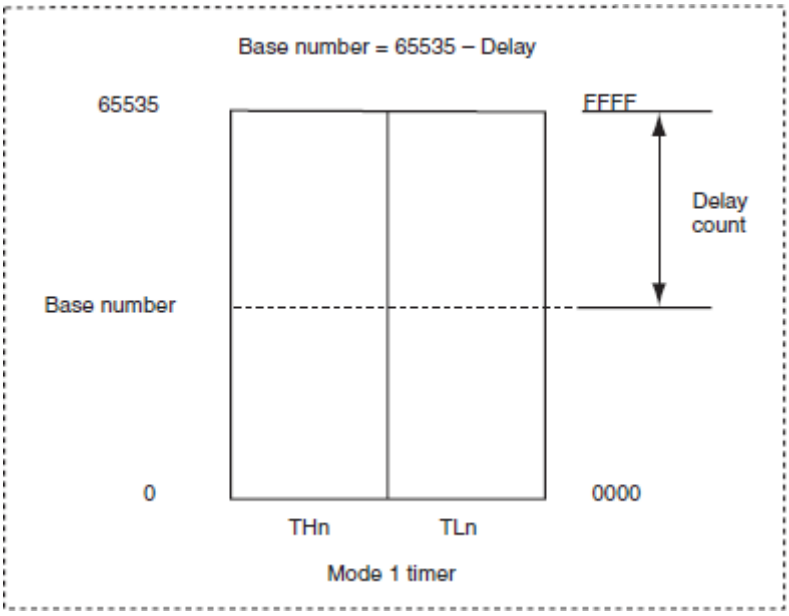


Figure 4.2 Determination of delay count for a mode 1 timer

Example 4.1 A 1 kHz square-wave signal is to be generated from pin 7 on port 1. The microcontroller clock frequency is 11.0592 MHz.

(a) Determine the required delay time.

(b) Using timer 0 determine the base numbers that must go into TH0 and TL0.

Solution

The required waveform is shown in Figure 4.3.

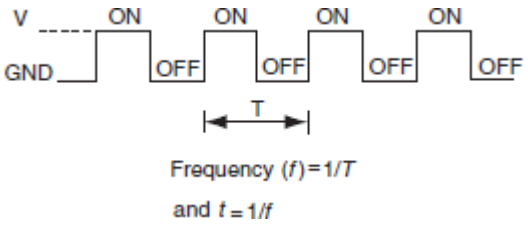


Figure 4.3 Square-wave signal to be generated at port 1, pin 7

(a) One cycle time T of the required square-wave signal equals

1/frequency T=1/1000 Hz = 1ms

Delay time = T/2 = 0.5ms

(b) Timer clock = microclock/12

= 11.0592MHz/12 = 0.9216 Mhz

Timer cycle time = 1/0.9216MHz =

1.0851μs Delay count = (delay time) /

(timer cycle time)

= 0.5ms / 1.0851 μs =460.8 ≅ 461 (nearest whole

number) Base number = 65535 – delay count

= 65535 – 461 = 65074

*65074 decimal = FE32 hexadecimal or a different

way, TH0 = whole number of 65074/256

65074/256 = 254.1953 = whole number of

254 TH0 = 254 assembly language MOV

TH0, #0FEH TL0 = (remainder of 65074/256

)x256 0.1953x256 = 49.9968 ≅ 50

TL0 = 50 decimal and assembly language MOV TL0, #32H

```
        MOV TMOD,       ; put Timer 0 into mode 1
STAR    #01H            ; set pin 7 on port 1 to
```

```
                ACALL DELAY
                CLR P1^7                ; clear pin 7 on port 1 to logic 0
                ACALL DELAY
                JMP START
DELAY:          MOV TH0, #0FEH          ; move hex FE into TH0
                MOV TL0, #32H           ; move hex 32 into TL0
                SETB TR0                ; turn timer 0 on
FLAG:           JNB TF0, FLAG           ; jump to FLAG if TF0 is not 1
                CLR TR0                 ; turn timer 0 off
                CLR TF0                 ; clear TF0 to zero
                RET                     ; return from subroutine
                END                     ; no more assembly code
```

Exercise 4.1 Repeat Example 4.1 to produce a 5 kHz square wave at port 1, pin 3. Assume the clock frequency remains at the same value.

Example 4.2 AT89C52 microcontroller having an 11.0592 MHz clock is to be used to generate a 1 kHz square-wave signal from pin 7 of port 1. Write a C program to achieve this.

Solution

A suitable program would be:

```
#include <reg51.h>                      // include 8051 register
#define on 1
#define off 0
sbit output = P1^7;                     // decleare a variable type sbit for P1^7
        void                            //delay_ms() returns nothing and takes
        delay_ms();
        main () {

                                        // timer1 : Gate=0 CT=0 M1=0 M0=0
                                        // timer0 : Gate=0 CT=0 M1=0 M0=1,
                                        Mod1
                while (1) {             // do for ever
                        output =        // P1.7 set to 1
                        1;              // wait for on time
                        delay_ms        // P1.7 set to 0
                        (); output =    // wait for off time
                        0;
                        delay_ms
                        ();
                                `
            TH0 =~ (461/256); // ~ complement operator
            TL0 =- (461%256);
            TR0 = on; // set TR0 of TCON to run timer0
            while (!TF0);       // wait for timer0 to set the Flag
```

```
        TR0 = off; // stop the timer0
        TF0 = off; // clear flag TF0
    }
```

MODE 2 EIGHT-BIT UP-COUNTER

The instruction **MOV TMOD, #02H** would put timer 0 into mode 2 defining an 8-bit timer using TL0 as the working register and TH0 as the automatic reload register. The circuit arrangement is shown in Figure 4.4.

In mode 2, the working register is only 8 bits wide and so the base number is 8 bits wide. When the TL register rolls over it is automatically reloaded with the contents of the TH register, whose loaded contents remain the same, so the base number goes into the TH register.

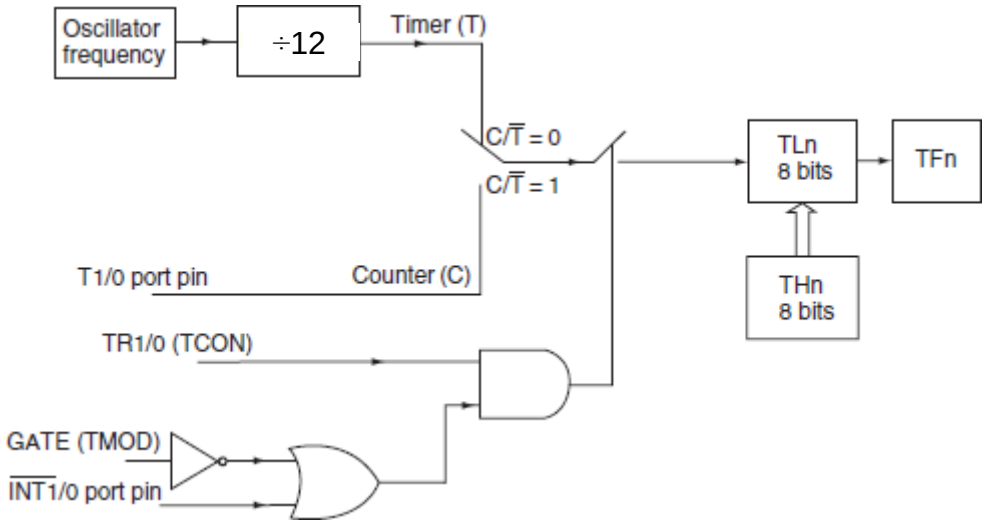


Figure 4.4 Circuit for timer 1/0 to operate as an 8-bit up-counter in mode 2

Example 4.3 AT89C52 microcontroller having an 11.0592 MHz clock is to be used to generate a 5 kHz square-wave signal from pin 7 of port 1. Write a suitable program to achieve this.

Solution

□□□□□□ − □□□□ □□□□□ $\frac{1}{5\square\square\ \square}$ = 0.2 □□
□□□□ =

□□□□□ □□□□□□□□ □□□ □ □□□□□□ □□□□ = *h*□□□ □*h*□ □□□□□ □□□□ = 0.1 □□ = 100μ□

□□□□□ 0 $\frac{\square\square\square\square\square\square\square}{\square\square\square} = \frac{11.0592\ \square\square\square}{12}$
□□□□□ =

12

□□□□□ = $\frac{12}{11.0592\ \square\square\square}$

□□□□□ 0 □□□□□

$$= 1.0851\ \mu\square$$

$$\square\square\square\square\square = \frac{\square\square\square\square\ \square\square\square\ \square}{\square\square\square\ \square\square\ \square\square\square\square\ \square\square\square\square\square\ \square\square\square} = \frac{100\mu\square}{1.0851\mu} = 92.1574\ \square$$

□□□□ 2 □□□□□ □□□□ □□□□□□ = 255 −

□□□□□ □□□□□ (255 = □□□□□□□ □□□□□ □□

8 − □□□ □□□□□□□□)

= 255 − 92.1574 = 162.8426 ≅ 163 (□□ □□□□□□□ □*h*□□ □□□□□□)

= □3 *h*□□

A3 hex is to go into

TH0 A3 hex is to go

into TL0

TL0 could start with its default value of 00H since the first half cycle would not be seen on an oscilloscope screen! The line MOV TL0, #0A3H can be left out of the program since after the first half cycle TL0 will automatically be reloaded with A3H from TH0.

```
        ORG 30H              ; program start address at 0030H
        MOV TMOD,            ; put Timer 0 into mode 2
        #02H MOV TH0,        ; auto-reload base number into TH0
START:  #0A3H SETB           ; set pin 7 on port 1 to logic 1
        P1^7
        ACALL DELAY
        CLR P1^7             ; clear pin 7 on port 1 to logic 0
        ACALL
        DELAY JMP
DELAY:  START SETB           ; turn timer 0 on
        TR0
FLAG:   JNB TF0, FLAG        ; jump to FLAG if TF0 is not 1
        CLR                  ; turn timer 0 off
        TR0                  ; clear TF0 to zero
        CLR                  ; return from subroutine
        TF0                  ; no more assembly code
        RET
        END
```

***Actually, we have to choose base number bigger than 163. Why?** (Try 0ADH)

Exercise 4.2 Repeat Example 4.2 to produce a 8 kHz square wave at port 1, pin 5. Assume the clock frequency remains at the same value.

EXPERIMENT 5: The 8051 Interrupts

TIMER INTERRUPT

When an interrupt occurs the processor pauses, saves the current program counter (PC) value into RAM designated by the stack pointer (SP) and then jumps to the interrupt vector address. The processor then carries out the instructions at the interrupt vector address and returns to the original program sequence, retrieving the previous PC data. The interrupt program must end with RETI (return from interrupt).

The AT89C52 has a total of six interrupt vectors: two external interrupts ($\overline{INT0}$ and $\overline{INT1}$), three timer interrupts (Timers 0, 1, and 2), and the serial port interrupt. These interrupts are all shown in Figure 5.1.

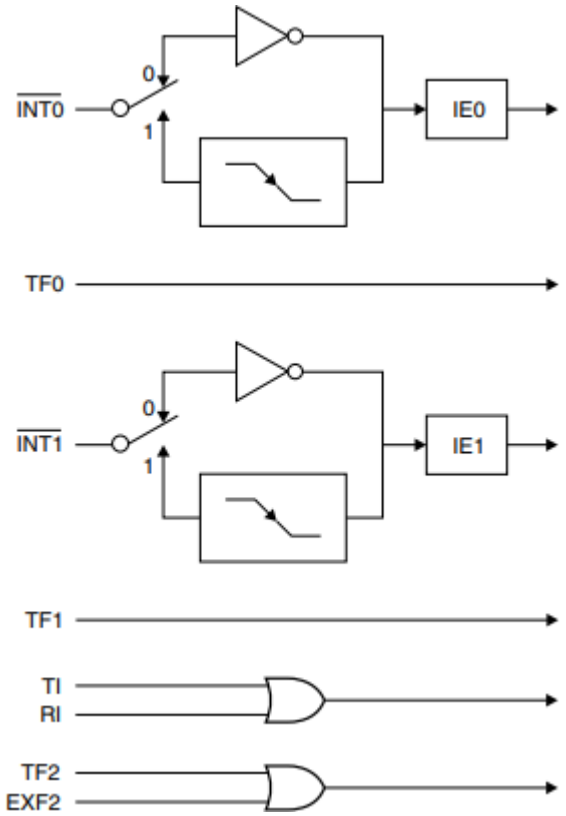


Figure 5.1 AT89C52 interrupt sources.

Each of these interrupt sources can be individually enabled or disabled by setting or clearing a bit in Special Function Register IE. IE also contains a global disable bit, EA, which disables all interrupts at once.

Interrupt Enable (IE) Register

EA	-	ET2	ES	ET1	EX1	ET 0	EX0
MSB						LSB	

Table 5.1 AT89C52 interrupt enable register

Symbol	Position	Function
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	Reserved.
ET2	IE.5	Timer 2 interrupt enable bit.
ES	IE.4	Serial Port interrupt enable bit
ET1	IE.3	Timer 1 interrupt enable bit.
EX1	IE.2	External interrupt 1 enable bit.
ET0	IE.1	Timer 0 interrupt enable bit.
EX0	IE.0	External interrupt 0 enable bit.

Table 5.2 AT89C52 interrupts

Source	Interrupt vector address
P3^2 / Int0	03H
Timer 0	0BH
P3^3 / Int1	13H
Timer 1	1BH
Serial Rcv.	23H
Serial Xmit.	23H
Timer 2	2BH
P1^1 / T2EX	2BH

It can be seen from Table 5.2 that previous assembly language programs started from address 0030H in order to leave the interrupt vectors as a reserved space.

Timer interrupts can be made to occur when the TF is set at rollover. This is achieved by setting the relative bits in the interrupt enable (IE) registers.

Example 5.1 Modify the program of the previous example such that a timer 0 interrupt causes the logic level on pin 7 port 1 to be toggled (switched to opposite logic level) producing a square wave of frequency 5 kHz.

Solution

Program

```
        ORG 00H          ; reset addresses
        SJMP             ; short jump over reserved area
        START            ; Timer 0 interrupt vector address
        ORG 0BH          ; go to interrupt routine
        SJMP             ; program start address at 0030H
        TASK
        ORG 30H
```

```
START:    MOV TMOD, #02H      ; put Timer 0 into mode 2
          MOV TH0,            ; auto-reload base number into TH0
          #0A3H SETB          ; enable all
          EA
```

```
                SETB ET0            ; enable Timer 0 interrupt
                SETB TR0            ; turn Timer 0 on
AGAIN:          SJMP AGAIN          ; stay here till interrupt occurs
TASK:           CPL P1^7            ; complement (i.e. toggle) pin 7 port
                                    1
                RETI                ; return from interrupt routine
                END                 ; end of assembly language
```

Example 5.2 Modify the C program of Example 4.2 such that a timer 0 interrupt causes the logic level on pin 7 port 1 to be toggled (switched to opposite logic level) producing a square wave of frequency 5 kHz.

Solution

```
#include <reg51.h>                  // include 8051 register file
#define on 1                        // on = 1
#define off 0                       // off = 0
sbit output = P1^7;                 // decleare a variable type sbit for
        main () {
                TMOD =              // put Timer 0 into mode 2
                0x02; TH0 =         // auto-reload base number into
                0xA3;               TH0
                EA =                // enable all
                on; ET0             // enable Timer 0 interrupt
                = on;               // turn Timer 0 on
                TR0 =
                on;
                while (1) {
                        }
               output = ~output; // toggle the
        }
```

Timer 2

The previous program could be viewed as the basis of a simple multitasking system where the microcontroller performed a task of complementing pin 7 on port 1 every 100 ms. The auto- reload and automatic clearing of the Timer Flag meant that once the timer reload register had been set up and the timer turned on it could be left to continually interrupt every 100 ms. Because the working register TL0 is only 8 bits wide the time duration of the interrupt signal is small.

The AT89C52 has Timer 2, which has 16-bit auto-reload giving a maximum count of 65535. Timer 2 has three operating modes:

1. capture mode
2. 16-bit auto-reload mode
3. baud rate generator mode.

Auto-reload is the default mode. Capture mode causes data in TL2 and TH2 to be transferred to the capture registers RCAP2L and RCAP2H when there is a 1-to-0 transition on T2EX (port 1.1).

Timer 1 can be used as the serial port baud rate generator but has limitations on the minimum baud rate. For example, with an oscillator frequency of 11.0592MHz the baud rate generation can only go down to the standard rate of 4800, for lower values the oscillator frequency must be lowered. Timer 2 having 16-bit auto-reload gets over this problem.

The control register associated with timer 2 is

T2CON. **Table 5.3** T2CON – Timer/Counter 2

Control Register T2CON Address = 0C8H

Bit addressable

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/RL2
7	6	5	4	3	2	1	0

MSB LSB

Symbol	Function
TF2	rollover or overflow flag, it must be cleared by software
EXF2	external flag set to 1 when there is a negative transition on port 1.1 (T2EX)
RCLK	when 1, baud rate generator mode
TCLK	when 1, baud rate generator mode
EXEN2	external enable flag, when 1 allows capture or reload following a negative transition on T2EX
TR2	turns timer 2 on (1) or off (0)
C/$\overline{T2}$	increments TL2, TH2 by onboard timer (0) or external negative edge on port1.0 (T2)
CP/RL2	when 1 (capture mode), when 0 (auto-reload on rollover)

Example 5.3 AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Write a program that causes timer 2 to generate an interrupt every 10 ms toggling pin 7 on port 1.

Solution

□□□□□□□□□□ □□□□□□□□□□ = 11.0592 □□□

□*h*□□□□□□□ □□□□□ □□□□□ $\frac{11.0592 \text{ □□□}}{12}$ =

□□□□□ □□□□□ □□□□□ =

1.0851μ□

□□□□□ □□□□ $\frac{10□□}{1.0851μ}$ = 9216 (□□□□□□□ □*h*□□ □□□□□□)

□□□□□ =

□

□□□□□ 2 □□□□ □□□□□□ = 65535 − 9216 = 56319 □□□□□□□ = □□□□□

In timer 2 the reload register for TH2 is RCAP2H (capture register 2) and the reload register for TL2 is RCAP2L. ET2 in IEN1 enables timer 2 interrupt.

Table 5.4 AT89C52 SFR Map for RCAP2H and RCAP2L

T2CON	T2MOD	RCAP2L	RCAP2H	TL2	TH2	-	-
0C8H	0C9H	0CAH	0CBH	0CCH	0CDH	0CEH	0CFH

The evaluation version of the assembly language software does not have timer 2 SFRs; therefore the program starts by equating (EQU) the SFR labels to their hex addresses.

The program uses OR logic (ORL) to force logic 1 in the SFRs without affecting other bits, and it also uses AND logic (ANL) to force logic 0.

Program:

```
RCAP2H      EQU 0CBH          ; sfr address = CBH
RCAP2L      EQU 0CAH          ; sfr address = CAH
IEN1        EQU 0A8H          ; sfr address = A8H
T2CON       EQU 0C8H          ; sfr address = C8H
            ORG 00H           ; reset addresses
            SJMP              ; short jump over reserved area
            START ORG         ; Timer 2 interrupt vector address
            2BH SJMP          ; go to interrupt routine
            TASK ORG          ; program start address at 0030H
START:      30H               ;
            MOV     RCAP2H,   ;
            #0FFH       MOV   ; enable all
            RCAP2L,   #0DBH   ; enable Timer 2 interrupt
            SETB EA           ; turn Timer 2 on
AGAIN:      ORL IEN1,         ; stay here till interrupt occurs
            #20H ORL
            T2CON, #04H
            SJMP AGAIN
TASK:       CPL P1^7          ; complement (i.e. toggle) pin 7 port
            ANL T2CON,        1
            #7FH RETI         ; clear Timer2 flag(TF2)
                              ; return from interrupt routine
            END               ; end of assembly language
```

Example 5.4 AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Write a C program that causes timer 2 to generate an interrupt every 50 ms toggling pin 7 on port 1.

Solution

□□□□□□□□□□ □□□□□□□□□□ = 11.0592 □□□

□*h*□□□□□□□ □□□□□ □□□□□ = $\frac{11.0592□□□}{12}$

□□□□ □□□□□ □□□□□ = 1.0851μ□

□□□□□ □□□□ □□□□□ = $\frac{50□□}{1.0851\mu□}$ = 46079

□□□□□ 2 □□□□ □□□□□□□ = 65535 − 46079 = 19456 □□□□□□□ = 4□00 □

TH2=4CH or TH2=76 decimal, TL2=00H or TL2=0 decimal.

```
#include <reg52.h>                  // include 8052 register file
sbit output =                       // decleare a variable type sbit for
P1^7; main () {
                T2CON =             // turn Timer 2 on
                0x04;               // put Timer 0 into mode 2
                RCAP2H=76           //auto-reload base number into
                ; RCAP2L=0;         TH0
                TL2 =
                RCAP2L;
                TH2 =               // Enable Timer 2 Interrupts
                RCAP2H;             // Start Timer 2 Running
                ET2 = 1;            // Global Interrupt Enable

                        while (1)   // do for
                        {
                                }

void timer2 (void) interrupt 5 {
                TF2=0;              // Clear the interrupt
                request output = ~output;  // toggle the output
}
```

External interrupt

Negative edge transitions on PORT 3 pins 2 ($\overline{INT0}$) and 3 ($\overline{INT1}$) can cause interrupts; their interrupt vector addresses are 03H and 13H respectively. A possible circuit arrangement is shown in Figure 5.2. Figure 5.2 shows a switch circuit where the voltage on P3.3 (port 3 pin 3) is normally 5 V. Pressing the switch causes a negative edge transition as the voltage switches down from 5V to 0 V. If the switch is pressed and held then a logic 0 level is held on P3.3. These are the two external interrupt conditions; it can be either edge triggered (transition logic 1 to 0) or level triggered (logic 0).

Interrupt	Port and pin	Interrupt vector address
$\overline{□□□□}$	P3^2	3H
$\overline{□□□□}$	P3^3	13H

The four least significant bits of the TCON register are used to set the external interrupt parameters.

TCON

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
MSB							LSB

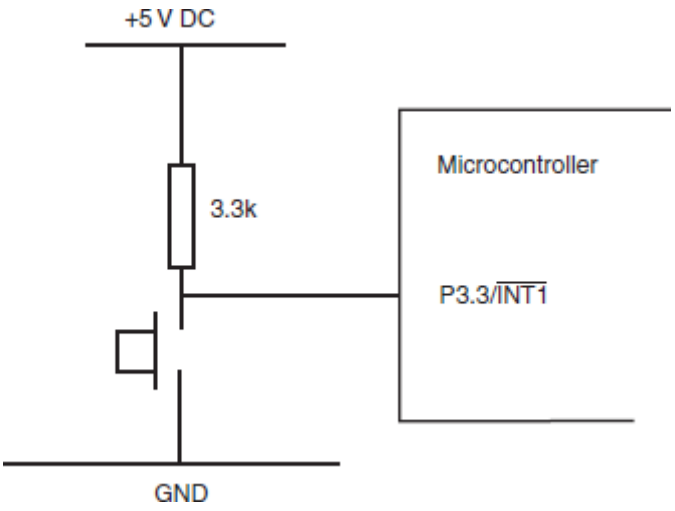


Figure 5.2 Circuit arrangement to produce an interrupt on port 3, pin 3. Asimilar arrangement can be used for port 3, pin 2

IT1=1, $\overline{INT1}$(P3.3) interrupt activated on a negative edge transition. IE1 flag set to 1 when there is a negative edge transition on $\overline{INT1}$(P3.3), cleared automatically when servicing the interrupt. IT1=0, $\overline{INT1}$(P3.3) interrupt activated on logic 0 level. IE1 flag set to 1 when there is a logic 0 level on INT1 (P3.3), cleared when the logic level on P3.3 goes back to logic 1.

The process for IT0 and IE0 is similar to the above.

Example 5.5 A AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Write an assembly language program that complements the logic level on port 1 pin 7 when an edge triggered interrupt occurs on port 3 pin 3 ($\overline{INT1}$).

Solution

```
        ORG 00H          ; reset addresses
        SJMP START       ; short jump over reserved area
        ORG 13H          ; INT1 interrupt vector address
        SJMP TASK        ; go to interrupt routine
        ORG 30H          ; program start address at 0030H
START:  SETB IT1         ; interrupt edge triggered
        SETB EA          ; enable all set interrupts
        SETB EX1         ; enable INT1 interrupt
AGAIN:  SJMP AGAIN       ; stay here till interrupt occurs
TASK:   CPL P1^7         ; complement (i.e. toggle) pin 7 port
                         1
        RETI             ; return from interrupt routine
```

```
END                                    ; end of assembly language
```

Interrupt Enable (IE) Register

EA	-	ET2	ES	ET1	EX1	ET 0	EX0
MSB						LSB	

Example 5.6 A AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Write a C program that complements the logic level on port 1 pin 7 when an edgetriggered interrupt occurs on port 3 pin 3 (INT1).

Solution

```
#include <reg52.h>                  // include 8052 register file
sbit output =                       // decleare a variable type sbit for
P1^7; main () {
        IT1 =                        // interrupt edge
        1;                           triggered
        EA = 1;                     // Global Interrupt Enable
        EX1

            while (1)               // do for
            {
                }

void ex0 (void) interrupt 2 {
        output = ~output;           // toggle the
        output
```

EXPERIMENT 6: Universal Asynchronous Receive Transmit (UART)

UARTs are used for serial communication between systems; they can be either half duplex (send or receive) or full duplex (send and receive at the same time). Also known as an RS232 connection the microcontroller UART can provide the connection with a PC or another microcontroller-based system. Figure 6.1 illustrates possible connection arrangements. In a minimum connection there could be only two transmission lines, transmit (Tx) and receive (Rx) as shown in Figure 6.2 The data is conveyed as a bit stream, either transmit or receive, and the speed is defined by the baud rate i.e. the bits per second.

The UART has four modes of operation, 0 to 3. Modes 0 and 2 have fixed baud rates, mode 0 is one-sixth of the oscillator frequency, and mode 2 is 1/16 or 1/32 of the oscillator frequency. For modes 1 and 3 the baud rate can be selected, a typical range is:

75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400

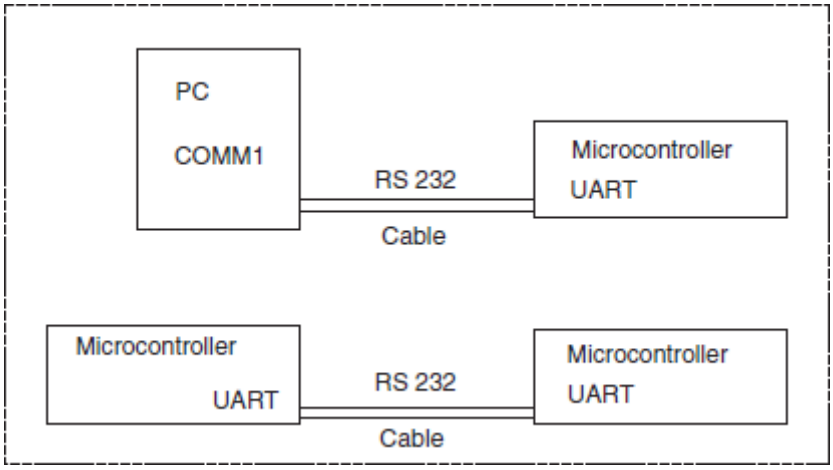


Figure 6.1 Use of RS232 interface between PC and microcontroller or between two microcontrollers

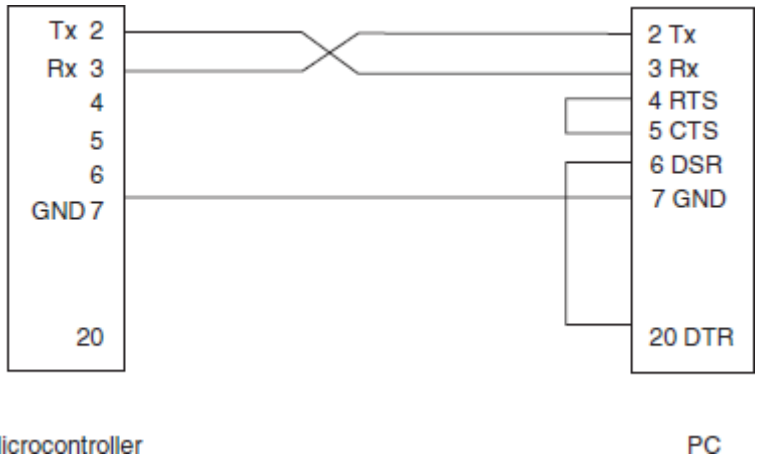


Figure 6.2 RS 232 transmit (Tx) and receive (Rx) connections between a PC and microcontroller

Modes 0 and 1 are used for connection between two devices. Modes 2 and 3 are used for master slave multiprocessor systems, in principle there could be one master microcontroller and up to 255 slave microcontrollers.

In mode 1 ten bits are used to specify an RS232 frame consisting of 1 start bit (logic 0), 8 data bits and 1 stop bit (logic 1). For example the ASCII bit pattern 0100 0001 (hex 41) represents the character A and is transmitted as shown in Figure 6.3; least significant bit (LSB) first.

The baud rate is defined by using one of the onboard timers usually timer 1 in mode 2 and for the AT89C52 microcontroller, timer 2.

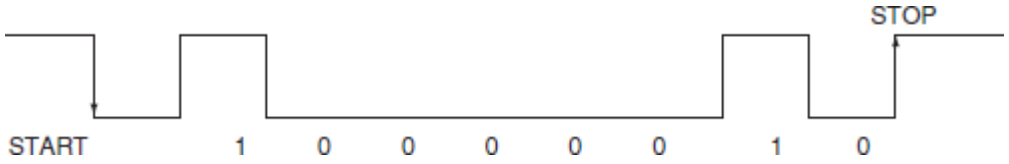


Figure 6.3 ASCII bit pattern 01000001 (41H) for character 'A'

SERIAL PORT CONTROL (SCON) REGISTER

SCON

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SM1	
0	0	Mode 0
0	1	Mode 1
1	0	Mode 2
1	1	Mode 3

SM2 when £=1, enables multiprocessor operation in modes 2

and 3 REN when £=1, enables serial reception

TB8 Used in multiprocessor operation in modes 2

and 3 RB8 Used in multiprocessor operation in

modes 2 and 3

TI Transmit interrupt flag, set when byte transmission is completed. Must be cleared by software

RI Receive interrupt flag, set when a byte in the serial buffer (SBUF) is ready for retrieval. Must be cleared by software.

BAUD RATE USING TIMER 2

Timer 2 is selected as the baud rate generator by setting TCLK and/or RCLK in T2CON. Note that the baud rates for transmit and receive can be different if Timer 2 is used for the receiver or transmitter and Timer 1 is used for the other function. Setting RCLK and/or TCLK puts Timer 2 into its baud rate generator mode, as shown in

Figure 6.4.

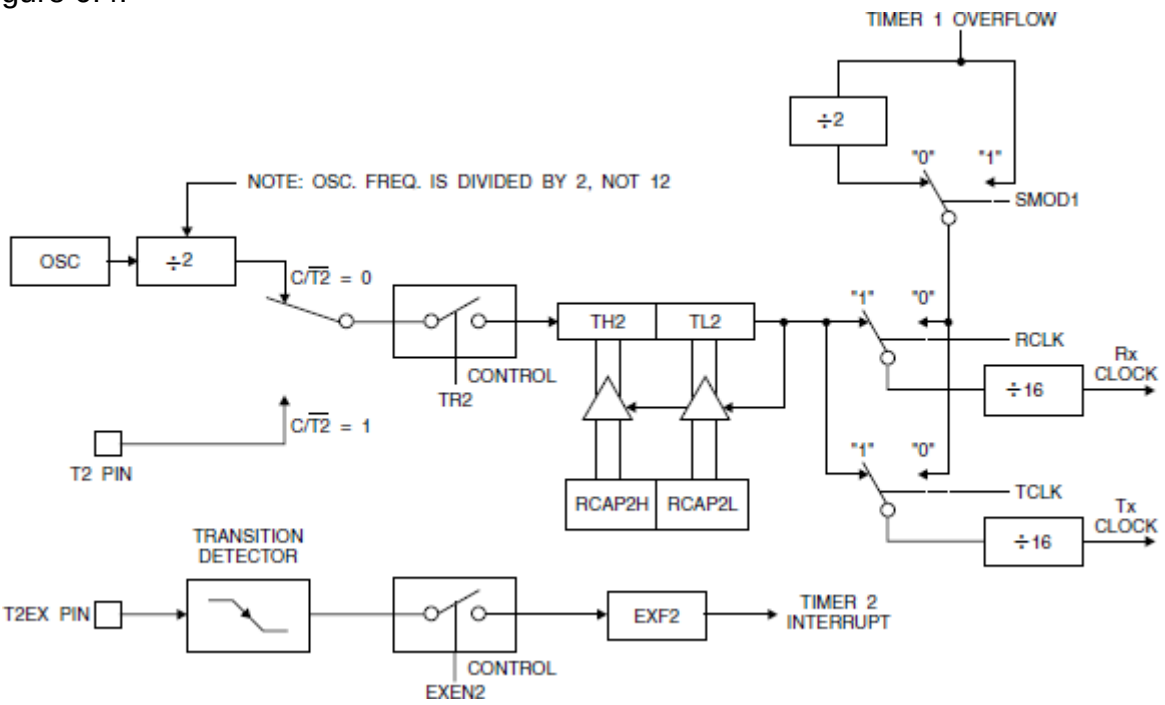


Figure 6.4 Timer 2 in Baud Rate Generator Mode

The baud rate generator mode is similar to the auto-reload mode, in that a rollover in TH2 causes the Timer 2 registers to be reloaded with the 16-bit value in registers RCAP2H and RCAP2L, which are preset by software.

The baud rates in Modes 1 and 3 are determined by Timer 2's overflow rate according to the following equation.

$$\text{Modes 1 and 3 Baud Rates} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

The Timer can be configured for either timer or counter operation. In most applications, it is configured for timer operation (CP/T2 = 0). The timer operation is different for Timer 2 when it is used as a baud rate generator. Normally, as a timer, it increments every machine cycle (at 1/12 the oscillator frequency). As a baud rate generator, however, it increments every state time (at 1/2 the oscillator frequency). The baud rate formula is given below.

$$\text{Modes 1 and 3 Baud Rates} = \frac{\text{Oscillator Frequency}}{32x[65536 - (\text{RCAP2H, RCAP2L})]}$$

$$(\text{RCAP2H, RCAP2L}) = 65536 - \frac{\text{Oscillator Frequency}}{32x(\text{Modes 1 and 3 Baud Rates})}$$

where (RCAP2H, RCAP2L) is the content of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

Timer 2 as a baud rate generator is shown in Figure 6.4. This figure is valid only if RCLK or TCLK
= 1 in T2CON. Note that a rollover in TH2 does not set TF2 and will not generate an interrupt.

Example 6.1 Assuming an oscillator frequency of 11.0592MHz and UART mode 1 determine the timer 2 capture values for baud rates of 38400, 19200, 9600, 300, 150.

Solution

$$(\text{RCAP2H}, \text{RCAP2L}) = 65536 - \frac{11059200\ \text{Hz}}{32x(38400)}$$

38400	RCAP2 = 65527 = FFF7H	RCAP2H = 0FFH	RCAP2L = 0F7H
19200	RCAP2 = 65518 = FFEEH	RCAP2H = 0FFH	RCAP2L = 0EEH
9600	RCAP2 = 65500 = FFDCH	RCAP2H = 0FFH	RCAP2L = 0DCH
300	RCAP2 = 64384 = FB80H	RCAP2H = 0FBH	RCAP2L = 80H
150	RCAP2 = 63232 = F700H	RCAP2H = 0F7H	RCAP2L = 00H

Example 6.2 A AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 2 and configuring the UART in mode 1 write an assembly language program that transmits the ASCII character A at a baud rate of 9600.

Solutio

n

Progra

m

```
SOCON       EQU 98H          ;SCON SRF address
SOBUF       EQU 99H          ;SBUF SRF address
RCAP2H      EQU 0CBH         ; sfr address = CBH
RCAP2L      EQU 0CAH         ; sfr address = CAH
T2CON       EQU 0C8H         ; sfr address = C8H
            ORG 00H          ; reset addresses
            SJMP             ; short jump over reserved area
            START            ; program start address at 0030H
START:      ORG 30H          ; serial mode 1, TI set
            MOV SOCON,       ; baudrate 9600
            #42H MOV
            RCAP2H, #0FFH    ; turn Timer 2 on
AGAIN:      MOV RCAP2L,      ; ASCII of A into S0BUF
            #0DCH ORL
```

```
                T2CON, #34H
                MOV SOBUF, # “A”




HERE:           JNB TI, HERE        ; stay here till TI set
                CLR TI              ; clear TI
                SJMP                ; repeat
                AGAIN               ; end of assembly language
                END
```

Example 6.3 A AT89C52 microcontroller has an oscillator frequency of 11.0592 MHz. Using timer 2 and configuring the UART in mode 1 write a C program that transmits the ASCII character A at a baud rate of 9600.

Solution

Program

```
#include                        // include 8052 register

main()                          // start of the program
                                // Serial mode 1
        SCON=0x4                // SM0 = 0, SM1 = 1, SM2 = 0,
                                // REN =0 TB8 = 0 RB8 = 0 TI = 0 RI=0
                                // Timer 1 in mode 2
        T2CON      =            // Baudrate = 9600
        0x04;                   //
        RCAP2H=0x               //
        FF;                     // Tx Clock flag,forces use of T2 in mode
        RCAP2L=0x               1, 3
        RC: TCLK =              // Start Timer 2 Running

            while (1)           // do for
                SBUF= 'A';      // load 'A' into Serial BUFFER
                while (!TI);    // wait for completion of
                transmission TI=0;  // clear transmission flag
                }
            }
```

SENDING A LINE OF TEXT

Example 6.4 Write an assembly language program that repeatedly sends the line of text 'ADU EEE Microprocessors'.

Solution

Program

```
SOCON       EQU 98H         ;SCON SRF address
SOBUF       EQU 99H         ;SBUF SRF address
            ORG 00H         ; reset addresses
            SJMP            ; short jump over reserved area
            START
            ORG 30H         ; program start address at 0030H
START:      MOV SOCON,      ; serial mode 1, TI set
            #42H MOV        ; timer 1 mode 2
            TMOD, #20H      ; baudrate 9600
            MOV TH1,        ; TL1 also initially set
            #0FDH MOV       ; turn Timer 1 on
TEXT:       TL1, #0FDH      ; Data Pointer to message address
            SETB TR1
            MOV DPTR, #MSG1
```

```
NEXTCH:         MOV A, #0                   ; zero the previous character
```

```
                MOVC A,          ; character into A
                @A+DPTR          ; checking end of message, ~=
                CJNE A, #7EH,    7EH
                TRXCH MOV A,     ; carriage return = 0DH
                #0DH             ; call up send routine
                ACALL            ; line feed = 0AH
                SEND MOV         ; call up send routine
                A, #0AH          ; repeat line of text
TRXC            ACALL            ; send text character
H:              SEND SJMP        ; increment data pointer
                TEXT             ; prepare to send next
                ACALL            character
                SEND INC         ; check SBUF clear to send
                DPTR SJMP        ; clear TI
                NEXTCH           : send contents of A
MSG1:           DB     'ADU EEE Microprocessors~' ; text message
                END              ; end of assembly
```

Example 6.5 Write a C program that repeatedly sends the line of text 'Our true mentor in life is science - M. Kemal ATATURK'.

Solutio

n

Progra

m

```
#include                        // include 8052 register
char message [] = “Our true mentor in life is science - M. Kemal
ATATURK~”;


void sendChar (char             // send one character
ch) {
                                // load character into Serial Buffer
        SBUF = ch;
        while                   // wait for completion of
                                transmission
        (!TI);
        TI=0;
        }
void sendMessage () {
        while (message [charPos] != ‘~’) {
                sendChar(message[charP
                os]); charPos++;
                }
        sendChar(0x0D);         // send carriage
        return
        sendChar(0x0A);         // send line feed
        }
```

```
main ()                         // start of the program
                                // Serial mode 1
        SCON=0x4                // SM0 = 0, SM1 = 1, SM2 = 0,
                                // REN =0 TB8 = 0 RB8 = 0 TI = 0 RI=0
        T2CON       =           // Timer 1 in mode 2
        0x04;                   // Baudrate = 9600
        RCAP2H=0x               //
        FF;                     // Tx Clock flag,forces use of T2 in mode
        RCAP2L=0x               1, 3
        RC: TCLK =              // Start Timer 2 Running


                while (1)       // do for
                        sendMessage
                        ();

        }
```

RECEIVING A CHARACTER

Example 6.6 Write an assembly language program that receives a character into the serial buffer (S0BUF) of the UART and writes the hex value of the character onto port 1. The character capture process is to start as the result of the UART being interrupted. The UART should be configured as mode 1.

Solution

The TI bit in the serial control (S0CON) can be left at its default value of zero but the receive bit (REN) must be set.

S0CON

SM0	SM1	SM2	REN	TB8	RB8	TI	RI
0	1	0	1	0	0	0	0

Thus S0CON = 50H. When the character byte is received RI will set but this must be cleared by the program to enable other receive interrupts to occur. In the simulation we should expect RB8 to set when the character byte has been received. TB8 and RB8 are mainly used in modes 2 and 3 but in mode 1 RB8 is set by the stop bit, which is the last bit of the 10-bit mode 1 data frame.

The interrupt enable (IEN0) register bits must be

set. IEN0

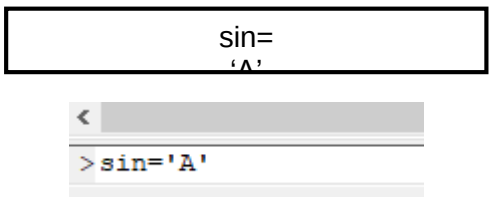
EA	EC	ES1	ES0	ET1	EX1	ET 0	EX0
1	0	0	1	0	0	0	0

Thus IEN0 = 90H. The UART interrupt vector address is at 0023H.

Program

```
S0CON       EQU 98H          ;SCON SRF address
S0BUF    EQU 99H             ;SBUF SRF address
IEN0        EQU 0A8H         ;interrupt SRF address
            ORG 00H          ; reset addresses
            SJMP             ; short jump over reserved area
            START            ; UART interrupt address
            ORG 23H          ; jump to interrupt routine
            SJMP             ; program start address at 0030H
            RXBUF
            ORG 30H
START:   MOV S0CON, #50H     ; serial mode 1, REN enabled
            MOV TMOD,        ; timer 1 mode 2
            #20H MOV TH1,    ; baudrate 9600
            #0FDH MOV        ; UART interrupt enabled
            IEN0, #90H       ; turn Timer 1 on
STAY:       SETB TR1         ; stay here, wait for interrupt
            SJMP STAY
RXBUF:      JNB RI, RXBUF    ; check for received byte
            CLR RI           ; clear RI
            MOV A,           ; move character from buffer to A
            S0BUF MOV        ; hex value onto port 1
            P1, A RETI       ; return from interrupt
            END              ; end of assembly language
```

- The following command injects the letter A into the simulated serial input.



Example 6.7 Write a C program that receives a character into the serial buffer (S0BUF) of the UART and writes the hex value of the character onto port 1. The character capture process is to start as the result of the UART being interrupted. The UART should be configured as mode 1.

Solutio

n

Progra

m

```
#include <reg52.h>                  // include 8052 register file
main () {                           // start of the program
                                    // Serial mode 1
        SCON=0x5                    // SM0 = 0, SM1 = 1, SM2
                                    // REN =1 TB8 = 0 RB8 = 0 TI = 0 RI=0
        T2CON      =                // Timer 1 in mode 2
        0x04;                       // Baudrate = 9600
        RCAP2H=0x                   //
        FF;                         // all interrupts enabled
        RCAP2L=0x                   // enable serial port
        DC; EA = 1;                 // Tx Clock flag,forces use of T2 in mode
        ES = 1;                     1, 3
        TCLK = 1;                   // Start Timer 2 Running

                while (1)           // do for
                {

    }
void SerialPort () interrupt 4 using
    P1=SBU                          // move received byte to port
    F;                              P1

}
```

EXPERIMENT 7: Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to.

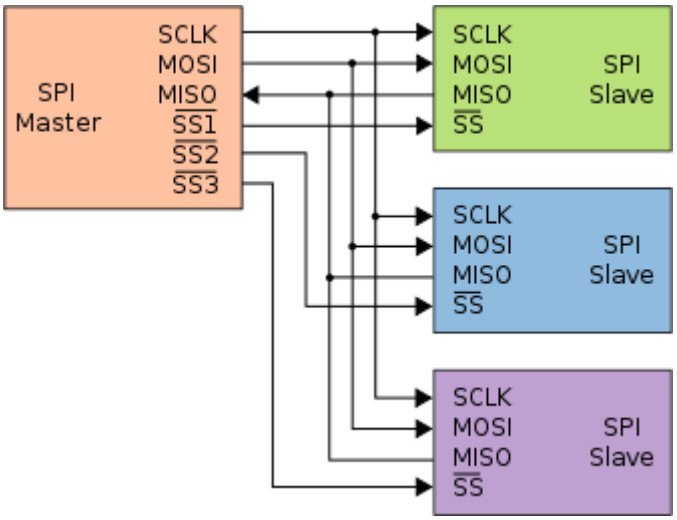


Figure 7.1 Typical SPI bus; master and three independent

slaves The SPI bus specifies four logic signals:

- SCK : Serial Clock (output from master)
- MOSI : Master Output Slave Input
- MISO : Master Input Slave Output
- SS : Slave Select (often active low, output from master)

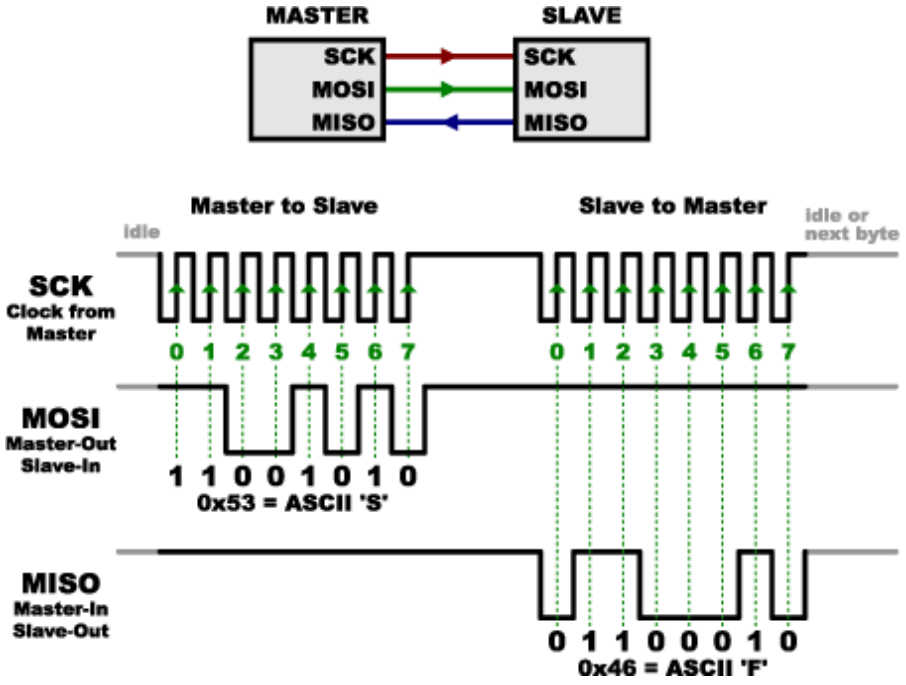
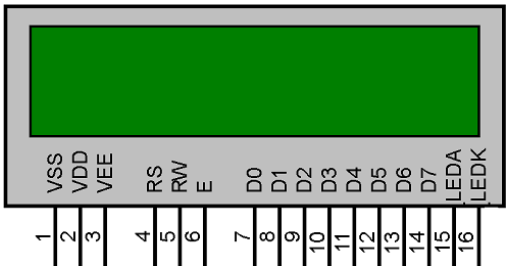


Figure 7.2 SPI signal example.

```
unsigned char SPI_transfer_byte(unsigned char byte_out)
{
        unsigned char byte_in = 0;
        unsigned char c_bit; //c:
        control

        for (c_bit = 0x80; c_bit == 0x00 ; c_bit >> 1 )
        {
                if ((byte_out & c_bit) == 1)
                {
                        MOSI = 1;
                }
                else
                {
                        MOSI = 0;
                }
                delay_ms(1
                ); SCK = 1;
                if (MISO == 1)
                {
                        byte_in |= c_bit;
                }
                delay_ms(1
                ); SCK = 0;
        }
        return byte_in;
}
```

PROJECTS

2x16 LCD DISPLAY



Pin Number	Symbol	Pin Function
1	VSS	Ground
2	VCC	+5V
3	VEE	Contrast adjustment
4	RS	Register Select. 0:Command, 1: Data
5	RW	Read/Write, R/W=0: Write & R/W=1: Read
6	E	Enable. Falling edge triggered
7	D0	Data Bit 0
8	D1	Data Bit 1
9	D2	Data Bit 2
10	D3	Data Bit 3
11	D4	Data Bit 4
12	D5	Data Bit 5
13	D6	Data Bit 6
14	D7	Data Bit 7/Busy Flag
15	LEDA	Back-light Anode(+)
16	LEDK	Back-Light Cathode(-)

Data Bus: As shown in the above figure and table, an alpha numeric lcd has a 8-bit data bus referenced as D0-D7. As it is a 8-bit data bus, we can send the data/cmd to LCD in bytes. It also provides the provision to send the the data/cmd in chunks of 4-bit, which is used when there are limited number of GPIO lines on the microcontroller.

Register Select(RS): The LCD has two register namely a Data register and Command register. Any data that needs to be displayed on the LCD has to be written to the data register of LCD. Command can be issued to LCD by writing it to Command register of LCD. This signal is used to differentiate the data/cmd received by the LCD.

If the RS signal is LOW then the LCD interprets the 8-bit info as Command and writes it Command register and performs the action as per the command.

If the RS signal is HIGH then the LCD interprets the 8-bit info as data and copies it to data register. After that the LCD decodes the data for generating the 5x7 pattern and finally displays on the LCD.

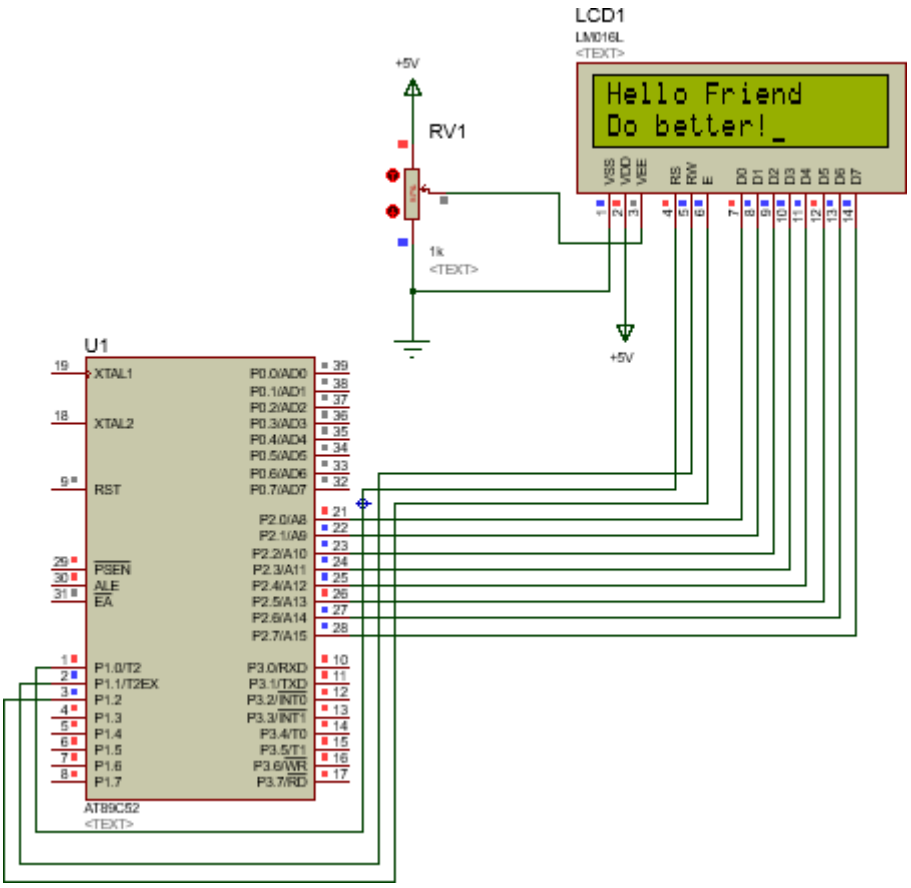
Read/Write(RW): This signal is used to write the data/cmd to LCD and reads the busy flag of LCD. For write operation the RW should be LOW and for read operation the R/W should be HIGH.

Enable(EN): This pin is used to send the enable trigger to LCD. After sending the data/cmd, Selecting the data/cmd register, Selecting the Write operation. A HIGH-to-LOW pulse has to be send on this enable pin which will latch the info into the LCD register and triggers the LCD to act accordingly.

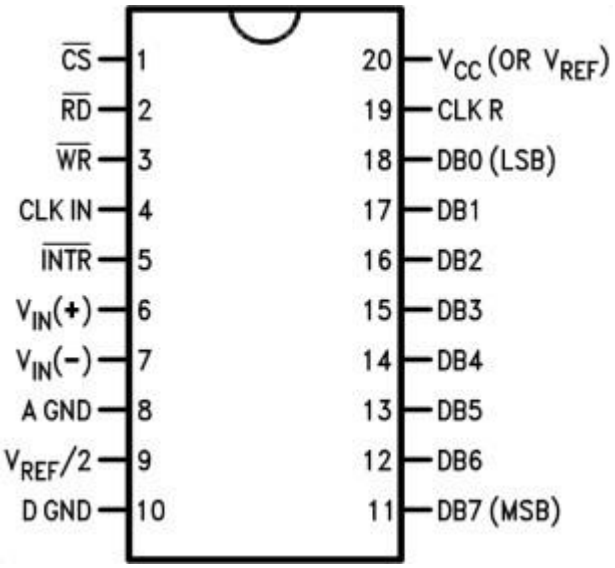
Example 1: Writing on the LCD screen

```
#include                                    // include 8052 register
//Configure the data bus and Control pins as per the hardware connection
//Databus is connected to P2_0:P2_7 and control bus P1_0:P1_2
#define LCD_Data_Bus P2

sbit LCD_RS =
P1^0; sbit
LCD_RW = P1^1;
sbit LCD_E =
P1^2;

int i,k;

//Local function to generate
delay void delay_function (int
d)
{
        for (i=0;i<d;i++);
}

//Function to send the command to LCD
void LCD_Command_Write (char Command){
        LCD_Data_Bus = Command; // Send the command to LCD
        LCD_RS = 0;                    // Select the Command Register by pulling RS
        LOW
        LCD_RW = 0;                    // Select the Write Operation by pulling RW
        LOW
        LCD_E  = 1;                    // Send a High-to-Low Pulse at
        Enable Pin delay_function(10);
        LCD_E = 0;
        delay_function(1000);
}
//Function to send the Data to
LCD void LCD_Data_Write
(char Data) {
   LCD_Data_Bus = Data;          // Send the data to LCD
   LCD_RS = 1;                   // Select the Data Register by pulling RS HIGH
   LCD_RW = 0;                   // Select the Write Operation by pulling RW
   LOW
```

```
int main () {
        char a, g[]=("Do better!");
        LCD_Command_Write(0x        // enable 5x7 mode for chars
        38);                        // Display OFF, Cursor ON
        LCD_Command_Write(0x        // Clear Display
        0E);                        // Move the cursor to beginning of first

        LCD_Data_Write('H');
        LCD_Data_Write('
        e');
        LCD_Data_Write('l'
        );
        LCD_Data_Write('l'
        );
        LCD_Data_Write('
        o');
        LCD_Data_Write('
        ');
        LCD_Data_Write('
        F');
        LCD_Data_Write('r'
        );
        LCD_Data_Write('i'
        );
        LCD_Data_Write('
        e');
        LCD_Data_Write('
}
```



Analog to Digital Converter (ADC0804)



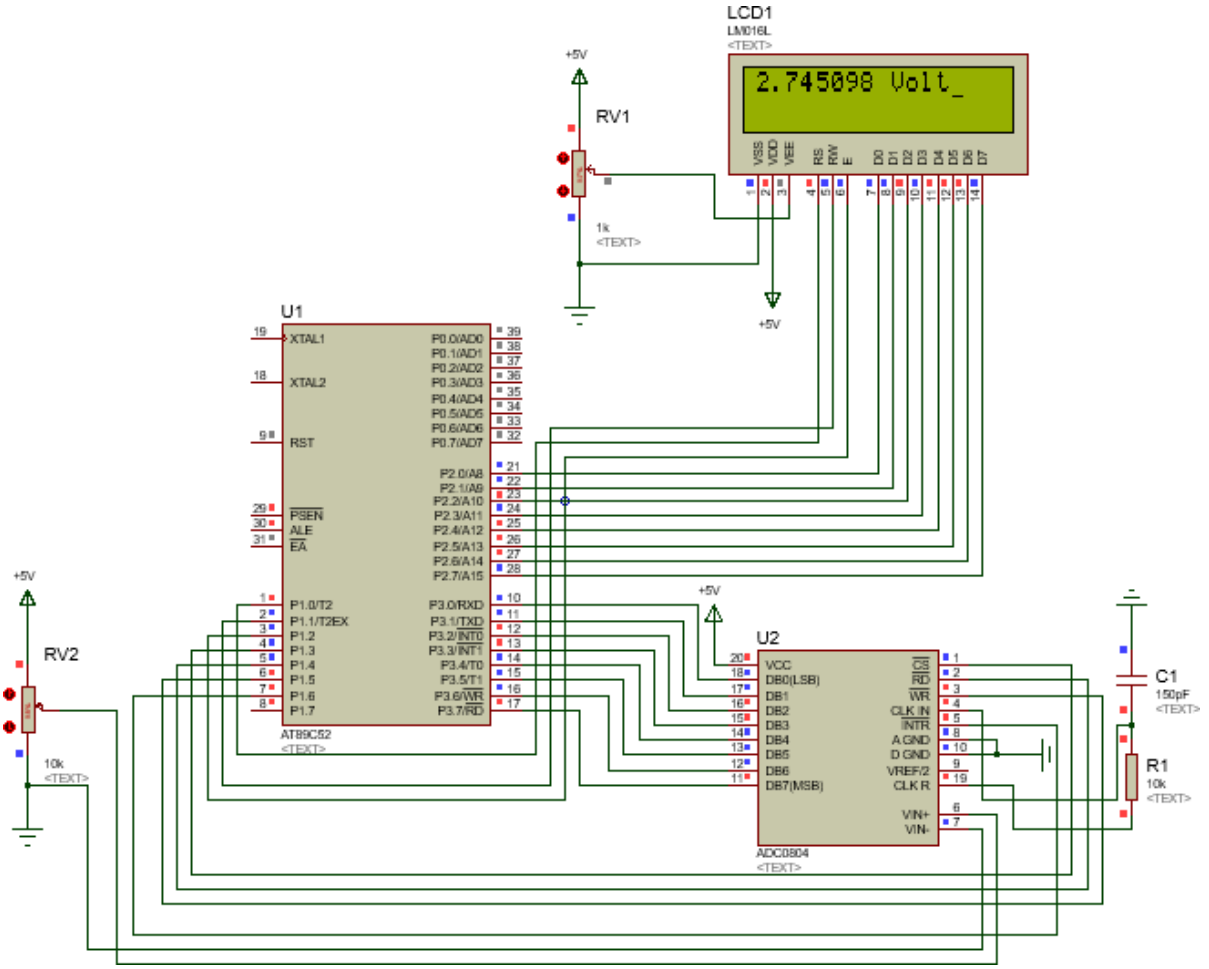
PIN		I/O	DESCRIPTION
NO.	NAME		
1	CS	I	Chip select
2	RD	I	Read
3	WR	I	Write
4	CLK IN	I	External Clock input or use internal clock generator with external RC elements
5	INTR	O	Interrupt request
6	Vin(+)	I	Differential analog input+
7	Vin(-)	I	Differential analog input–
8	A GND	I	Analog ground pin
9	$V_{REF}/2$	I	Reference voltage input for adjustment to correct full scale reading
10	D GND	I	Digital ground pin
11	DB7	O	Data bit 7
12	DB6	O	Data bit 6
13	DB5	O	Data bit 5
14	DB4	O	Data bit 4
15	DB3	O	Data bit 3
16	DB2	O	Data bit 2
17	DB1	O	Data bit 1
18	DB0 (LSB)	O	Data bit 0
19	CLK R	I	RC timing resistor input pin for internal clock generator
20	V_{CC} (or V_{REF})	I	+5V supply voltage, also upper reference input to the ladder

Example 2: Read a voltage value and print it on the LCD screen.

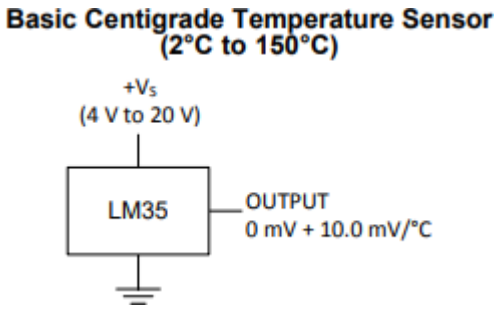
```
#include                        // include 8052 register file
<reg52.h>                       // include standard input
//Configure the data bus and Control pins as per the hardware
connection
//Databus is connected to P2_0:P2_7 and control bus P1_0:P1_2
#define LCD_Data_Bus P2
#define ADC_Data_Bus P3

sbit LCD_RS =
P1^0; sbit
LCD_RW = P1^1;
sbit LCD_E =
P1^2;

sbit    ADC_CS    =
P1^3; sbit ADC_RD
=      P1^4;      sbit
ADC_WR  =  P1^5;
sbit   ADC_INTR   =
P1^6;
```

```
//Local function to generate
delay void delay_function (int
d)
{
        for (i=0;i<d;i++);
}

//Function to send the command to LCD
void LCD_Command_Write (char Command){
        LCD_Data_Bus = Command; // Send the command to LCD
        LCD_RS = 0;                  // Select the Command Register by pulling RS LOW
        LCD_RW = 0;                 // Select the Write Operation by pulling RW LOW
        LCD_E  = 1;                 // Send a High-to-Low Pulse at
        Enable Pin delay_function(10);
        LCD_E = 0;
        delay_function(1000);
}
//Function to send the Data to
LCD void LCD_Data_Write
(char Data) {
        LCD_Data_Bus = Data;      // Send the data to LCD
        LCD_RS = 1;                 // Select the Data Register by pulling RS HIGH
        LCD_RW = 0;                 // Select the Write Operation by pulling RW LOW
        LCD_E  = 1;                 // Send a High-to-Low Pusle at
        Enable Pin delay_function(10);
        LCD_E = 0;
        delay_function(1000);
}
int main () {
        char buffer[16];

        LCD_Command_Write(0x38);         // enable 5x7 mode for chars
        LCD_Command_Write(0x0E);         // Display OFF, Cursor ON
        LCD_Command_Write(0x01);        // Clear Display
        LCD_Command_Write(0x80);        // Move the cursor to beginning of first
        line

        ADC_CS = 0;
        ADC_RD = 1;
        ADC_WR = 0;
        ADC_WR = 1;              //Low to high pulse to WR for starting conversion

        while (ADC_INTR);       // Wait until INTR =
        0 ADC_CS = 0;           //Ensure CS=0
        ADC_RD = 0;             //high to low pulse to RD for reading the data from
        ADC adc_data = ADC_Data_Bus;
        voltage = adc_data*5/255;
        sprintf(buffer,"%f %s",voltage,
        "Volt");
```

```
        for (k=0; buffer[k] != 0; k++)
        {
                LCD_Data_Write(buffer[k]);
        }
}
```



LM35 Temperature Sensor

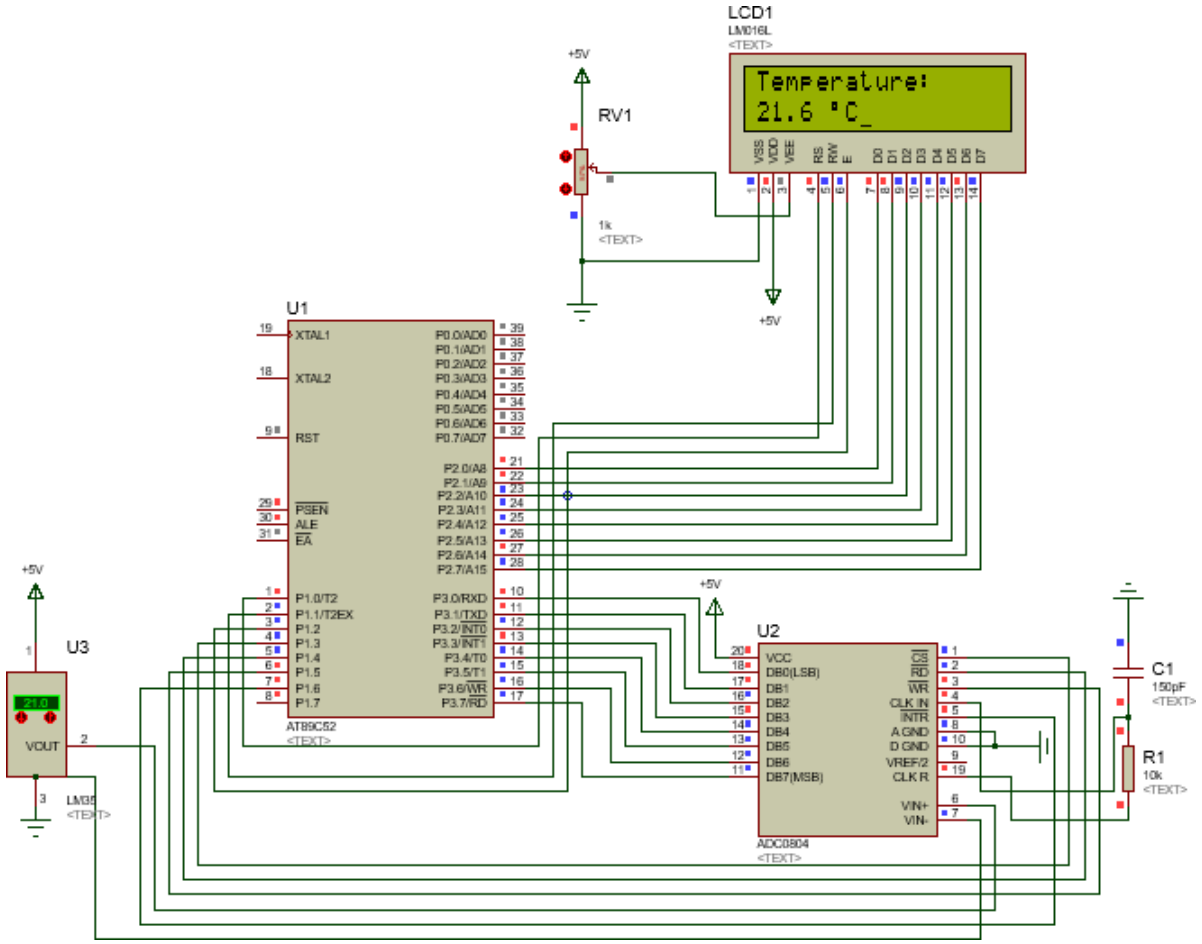



```
#include                    // include 8052 register file
<reg52.h>                   // include standard input
```

```
//Configure the data bus and Control pins as per the hardware connection
//Databus is connected to P2_0:P2_7 and control bus
P1_0:P1_2 #define LCD_Data_Bus P2
#define ADC_Data_Bus P3

sbit LCD_RS =
P1^0; sbit
LCD_RW = P1^1;
sbit LCD_E =
P1^2;

sbit    ADC_CS    =
P1^3;  sbit  ADC_RD
=       P1^4;       sbit
ADC_WR   =   P1^5;
sbit   ADC_INTR   =
P1^6;

int i,k;
float
adc_data=0;
float voltage=0;
float temperature=0;

//Local function to generate
delay void delay_function (int
d)
{
        for (i=0;i<d;i++);
}

//Function to send the command to LCD
void LCD_Command_Write (char Command){
        LCD_Data_Bus = Command; // Send the command to LCD
        LCD_RS = 0;                    // Select the Command Register by pulling RS LOW
        LCD_RW = 0;                    // Select the Write Operation by pulling RW LOW
        LCD_E  = 1;                    // Send a High-to-Low Pulse at
        Enable Pin delay_function(10);
        LCD_E = 0;
        delay_function(1000);
}
//Function to send the Data to
LCD void LCD_Data_Write
(char Data) {
   LCD_Data_Bus = Data;   // Send the data to LCD
   LCD_RS = 1;  // Select the Data Register by pulling RS HIGH
   LCD_RW = 0;    // Select the Write Operation by pulling RW
   LOW LCD_E  = 1;      // Send a High-to-Low Pusle at Enable
   Pin delay_function(10);
   LCD_E = 0;
   delay_function(1000);
```

```
}
int main () {
        char buffer[16];
```

```
        char T[] =
        ("Temperature:");
                                        // enable 5x7 mode for
LCD_Command_Write(0x0E);  // Display OFF, Cursor ON
LCD_Command_Write(0x01);  // Clear Display
LCD_Command_Write(0x80);  // Move the cursor to beginning of first
line

        ADC_CS = 0;
        ADC_RD = 1;
        ADC_WR = 0;
        ADC_WR = 1; //Low to high pulse to WR for starting conversion

        while (ADC_INTR); // Wait until INTR
        = 0 ADC_CS = 0; //Ensure CS=0
        ADC_RD = 0; //high to low pulse to RD for reading the data from
        ADC adc_data = ADC_Data_Bus;
        voltage = adc_data*5/255;
        temperature = voltage*1000/10; // Convert V to mV and
                                        // Divide 10 for temperature

        sprintf(buffer,"%4.1f                                   //223->for degree

        for (k=0; T[k] != 0; k++)
        {
                LCD_Data_Write(T[k]);
        }
        LCD_Command_Write(0xc0); //Go to Next
        line

        for (k=0; buffer[k] != 0; k++)
        {
                LCD_Data_Write(buffer[k]);
        }
}
```



Switching the device on and off with Bluetooth-controlled relay

```
#include
<reg52.h> sbit
out = P1^0;

char SerialRead(); // Seri porttan karakter okuma fonksiyonu
void SerialBegin(int i); // Seri portun ayarlarini yapma ve baslatma
fonksiyonu void init(); // mikrodenetleyiciyi baslatma fonksiyonu

main ()
{
        init();
        SerialBegin(960
        0); while(1) {
                char c1 = SerialRead();
```

```
                if (c1 == 'm')
                {
                        out = 1;
                }
                if (c1 == 'w')
                {
                        out = 0;
                }
        }
}

void init()
{
        P0      =
        0x00;  P1
        =    0x00;
        P2      =
        0x00;  P3
        = 0x00;
}

char SerialRead(){
        char x =
        0; if
        (RI==1){
                x =
                SBUF;
                RI = 0;
        }
        return x;
}

void SerialBegin(int i)
{
        unsigned long int number = 65536-
        345600/i; int i2 = number/256;
        SCON = 0x50; // mod 1 iki yönlü haberlesme için SCON registeri ayarlandi
        T2CON = 0x30; // Timer 2 baud rate generator olarak ayarlandi --> hem Rx, hem de
//Tx
        RCAP2H = i2;
```

RC
AP
2L
=
nu
mb
er-
i2*2
56;

```
        TR2 = 1; //Timer 2 açildi
}
```

Distance measurement with using ultrasonic sensor

```
#include <reg52.h> //8052 register dosyasi dahil
edildi #include <stdio.h> //standart giris çikis dahil
edildi
#include <math.h> //matematiksel islemler için kütüphane dahil
edildi #define LCD_Veri_Yolu P0
sbit LCD_RS =
P1^0; sbit
LCD_RW = P1^1;
sbit LCD_E =
P1^2;
//Ultrasonik sensorün
baglantisi sbit Trigger_pin =
P1^3;
sbit Echo_pin =
P1^4; int i,k;
void timer_gecikme_ms(int ms) { //timer 0 kullanilarak gecikme
        olusturuldu for (i=0; i<ms;i++){ //1xms kadar gecikme
        olusturulabilir
                TH0 = 0xFC; //1ms gecikme
                olusturuldu TL0 = 0x66;
                TR0 = 1; // timer 0 açildi
                while(!TF0); //timer 0 bayragi 1 olana kadar
                bekle TR0 = 0; // timer 0'i kapat
                TF0 = 0; // timer 0 bayragini sifirla
        }
}
void timer_gecikme_us() { //10us gecikme
                fonksiyonu TH0 = 0xFF; //10.85us
                gecikme olusturuldu TL0 = 0xF5;
                TR0 = 1; // timer 0 açildi
                while(!TF0); //timer 0 bayragi 1 olana kadar
                bekle TR0 = 0; // timer 0'i kapat
                TF0 = 0; // timer 0 bayragini sifirla
}
void gecikme_fonksiyonu (int d)
{
        for (i=0;i<d;i++);
}
//LCD ekrana komutlari yazan
fonksiyon void LCD_Komut_Yaz (char
Komut){
        LCD_Veri_Yolu = Komut;   // Komutlar LCD'ye gönderildi
        LCD_RS = 0;              // RS 0 yapilarak komut registeri
```

```
        gecikme_fonksiyonu(1000);
}
//LCD'ye veri yazan
fonksiyon void
LCD_Veri_Yaz (char Veri) {
  LCD_Veri_Yolu = Veri;            // Veriler LCD ekrana
  gönderildi LCD_RS = 1;        // RS 1 yapilarak veri
  registeri secildi LCD_RW = 0; // Yazma islemi için RW 0
  yapildi
  LCD_E  = 1;             // Enable için 1-0 komutlari
  gönderildi gecikme_fonksiyonu(10);
        LCD_E = 0;
        gecikme_fonksiyonu(1000);
}
void
        Tetikleme_sinyali_gond
        er(){ Trigger_pin = 1;
        timer_gecikme_us ();
        Trigger_pin = 0;
}
int main () {
        float
        uzaklik,deger,zaman;
        char buffer[16];
        char buffer1[16] = "Uzaklik =
        "; TMOD = 0x01; //Timer 0
        mode 1

        while(1) {
                LCD_Komut_Yaz(0x38); // Karakterler için 5x7 modu seçildi
                LCD_Komut_Yaz(0x0E); // LCD ekran kapatildi, imleç açik
                LCD_Komut_Yaz(0x01); // LCD ekran silindi
                LCD_Komut_Yaz(0x80); // Imleç en basa alindi
                Tetikleme_sinyali_gonder();
                while(!Echo_pin
                ); TH0 = 0;
                TL0 = 0;
                TR0 = 1;
                while(Echo_pin &&
                !TF0); TR0 = 0;
                deger = (TL0+TH0*256.0);
                zaman = (1.0851*pow(10,-6)*deger)/2.0; //s cinsinden
                süre uzaklik = 34320*zaman; //34320 cm/sn cinsinden
                ses hizi
                sprintf(buffer, "%.2f", uzaklik); //float türünden uzaklik degerini char tipine
//dönüstürüyor
        for (k=0; buffer[k] != 0; k++)
        {
                LCD_Veri_Yaz(buffer[k]);
        }
        timer_gecikme_ms(2000);
```

```
}
}
```

References

[1] The 8051 Microcontroller Fourth Edition - I. Scott MacKenzie

[2] 8051 Microcontroller An Applications Based Introduction, Chris Braithwaite, Fred Cowan, Hassan Parchizadeh

[3] http://editsworld.com/8051-microcontroller-architecture-block-diagram/

[4] https://www.edsim51.com/

[5] http://www.keil.com/dd/docs/datashts/atmel/at_c51ism.pdf

[6] http://ww1.microchip.com/downloads/en/appnotes/s72052.pdf