Microprocessor Laboratory Manual

EE3751

Gabriel Augusto Marques Tarquinio de Souza

Department of Electrical and Computer Engineering
Louisiana State University
and
Agricultural and Mechanical College
Baton Rouge, LA 70803

Rev. 06/05

Administrative Policy of the Laboratory

1) You are not allowed to smoke, eat or drink in the Laboratory. You are expected to conduct yourself professionally, and to keep your bench area *clean* and *neat*. You are required to return all equipment and parts used in the experiment to their proper places before you leave the lab.
2) You are expected to work in a group, which is defined to contain *at most* two people. If there are an odd number of people in the class section you are enrolled to, then someone will work alone. You are allowed to select your partner, but you must select a partner who does not have the same major as you, in other words, if you are an EE student you have to select an ECE student to be your partner, and if you are ECE student you must select an EE student as a partner. In the case there aren't enough students of different majors in the same section then groups formed with student of the same majors will be allowed.
3) You are encouraged to work on your programs outside of the lab. You are also expected to *build* and *test* your circuits, which may be built in advance. It is your responsibility to demonstrate to your lab instructor that the circuits you built do what they are supposed to do!
4) Lab reports will consist of your program listing.
5) If in a particular lab session you finish your experiment ahead of time you may choose to work on any of the previous experiments you have attempted and were unable to get it to work. You will not be allowed to work on an experiment you have not worked on previously.
6) In order to be able to borrow the lab equipment you must sign the Equipment Usage Policy.
7) The tools and instruments in each lab station are not supposed to be removed from the laboratory.
8) The Instructors are available to assist you in debugging the software or troubleshoot the hardware, but it is not the Instructors' responsibility to make your experiment work. It is therefore the responsibility of the student to complete the lab.
9) There will be a make up lab session at the end of the semester. During that session you will be allowed to make up *at most* two experiments you have previously worked on. Note that this is an opportunity being afforded to you so that you may work again on an experiment you previously attempted but were unsuccessful. If you did not work on an experiment during a previous lab session you will not be allowed to make it up during the make up session.
10) There will be a one-hour final practical examination, consisting of a program you are expected to write, debug, and demonstrate that it works. Every student will work *alone* on the final exam.

Experiment 0 - Introduction to DEBUG and the Assembly Process

This experiment will introduce you to DEBUG and TASM, allowing you to become familiar with the process of assembling, debugging and executing an assembly language program with a PC. DEBUG is a program available with every version of WINDOWS and DOS. Here you will learn how to use DEBUG to assemble, disassemble, execute and debug assembly language programs with a PC. You will also be instructed on how to examine and modify the memory and CPU registers of your PC. TASM (Turbo Assembler) is the assembler you will use in the lab this semester. One may find many assemblers like: MASM, NASN, TASM, etc, on the web. I recommend you download one to your computer at home so that you may work on your project away from the lab. In this experiment you will copy, edit, assemble, link, debug and execute the a program. During this process you will be using several programs, such as Turbo Assembler, Turbo Linker, Debug, and Notepad or Edit to accomplish your task. The knowledge acquired in this experiment will be extremely useful when working with the programs you will write during the semester. You may want to modify and save the program so that you can use it as the starting point when programming you next assignments. Do that by eliminating the red italicized statements.

The following convention will be used with all examples shown in this manual:

CAPITALIZED ITALICS REPRESENTS THE INFORMATION TYPED BY THE USER.

CAPITALIZED BOLD REPRESENTS THE COMPUTERS RESPONSE.

It is also suggested that the student uses a floppy disk in drive A: to store their programs. This will avoid the possibility of having your programs copied in case you forget to erase them from drive C:. If you use the hard drive make sure you use the C:\WORK directory.

Write the answers to the questions in this experiment on a separate sheet of paper, and then hand the answer sheet to the Instructor before leaving the lab.

Loading DEBUG

1 – Open a DOS window by double clicking the DOS WINDOW icon on the desktop or in the Programs Menu.
2 – Issue the following command at the DOS prompt to load DEBUG:

A:*DEBUG <ENTER>*
-

Note that the prompt is a small dash.

Exiting DEBUG

3 – Issue the following command at the DEBUG prompt to exit DEBUG:

```
-Q<ENTER>
A:\
```

Examining and modifying the contents of registers

4 – Load DEBUG and at the DEBUG prompt enter the following command

```
-R<ENTER>
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0100   NV UP EI PL
NZ NA PO NC
0B2C:0100 7509          JNZ   010B
-R CX<ENTER>
CX 0000
:0008<ENTER>
-R<ENTER>
AX=0000  BX=0000  CX=0008  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0100   NV UP EI PL
NZ NA PO NC
0B2C:0100 7509          JNZ   010B
```

Compare your display with the one shown above and note any possible discrepancies.

```
-R CX<ENTER>
CX 0008
:321<ENTER>
-R<ENTER>
AX=0000  BX=0000  CX=0321  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0100   NV UP EI PL
NZ NA PO NC
0B2C:0100 7509          JNZ   010B
-
```

Note that DEBUG displays all numeric values as hexadecimal numbers and that if you enter a value smaller than 4 digits, i.e. 321, DEBUG will pad it with zeros, 0321, when it writes them to the register.

5 – Modify the contents of register DX to 1F54.

6 – Now modify the contents of register DL from 54 to 68 without modifying the contents of register DH which you set to 1F in the previous step. Show the Instructor the results of steps 5 and 6.

Assembling, disassembling, and executing programs

7 – Below you will find a series of commands to assemble and execute a program that adds the contents of registers AX and BX. Practice the procedure by entering the same information and verifying the results displayed. Note that the segment address may be different than 0B2C.

```
-A 100<ENTER>
0B2C:0100 MOV AX,1<ENTER>
0B2C:0103 MOV BX,2<ENTER>
0B2C:0106 ADD AX,BX<ENTER>
0B2C:0108 INT 3<ENTER>
0B2C:0109<ENTER>
-R<ENTER>
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0100   NV UP EI PL
NZ NA PO NC
0B2C:0100 B80100        MOV   AX,0001
-G<ENTER>
AX=0003  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0108   NV UP EI PL
NZ NA PE NC
0B2C:0108 CC            INT   3
-
```

You were exposed to two new commands:

A - used for the assembly of programs.

Format: A <starting address>.

The starting address may be given as an offset to the code segment address. This was done in the example above.

G - used for the execution of programs.

Format: G <=starting address> <ending address>.

Execution will start at CS:IP if no addresses are given. This was demonstrated in the example above.

8 – Shown below are two formats for the command used to disassemble the program given in the example above. The first format uses a beginning and an ending address, and the second format uses a beginning address and a count of the number of bytes to be disassembled. Notice that the count is preceded by an 'L'.

```
-U 100 108<ENTER>
0B2C:0100 B80100        MOV   AX,0001
0B2C:0103 BB0200        MOV   BX,0002
0B2C:0106 01D8          ADD   AX,BX
0B2C:0108 CC            INT   3
-U 100 L9<ENTER>
0B2C:0100 B80100        MOV   AX,0001
```

```
0B2C:0103 BB0200        MOV   BX,0002
0B2C:0106 01D8          ADD   AX,BX
0B2C:0108 CC            INT   3
-
```

9 – Write a program to subtract the content of register DX from the content of register AX, then add the result to the content of CX. Set the registers to 4, 0A and 1F respectively. Assemble, execute, verify the results of the execution of your program, then disassemble it and show the instructor the results obtained.

Tracing the execution of your program

10 – The trace command T is used to trace the execution of a program by displaying register information after the execution of the each instruction in the selected range.
Format: T <=starting address> <number of instructions>
Like the Go command if the starting address is not specified, it starts execution at CS:IP.

```
-T =100 4<ENTER>

AX=0001  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0103   NV UP EI PL
NZ NA PO NC
0B2C:0103 BB0200        MOV   BX,0002

AX=0001  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0106   NV UP EI PL
NZ NA PO NC
0B2C:0106 01D8          ADD   AX,BX

AX=0003  BX=0002  CX=0000  DX=0000  SP=FFEE  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0B2C  IP=0108   NV UP EI PL
NZ NA PE NC
0B2C:0108 CC            INT   3

AX=0003  BX=0002  CX=0000  DX=0000  SP=FFE8  BP=0000
SI=0000  DI=0000
DS=0B2C  ES=0B2C  SS=0B2C  CS=0590  IP=13B1   NV UP DI PL
NZ NA PE NC
0590:13B1 55            PUSH  BP
-
```

Check the example above then trace the execution of your program. Show the results to the instructor.

Accessing and modifying data in DEBUG

11 – In this section you will be exposed to the three commands: F – fill, D – dump, and E – enter. This command's address reference the data segment (DS). If you need to access information in another segment you need to include the segment in the address. The command's description and usage examples are given below:

F – used to fill blocks of memory with data.

Format: F <starting address> <ending address> <data>
D <starting address> <L number of bytes> <data>

D – used to display the memory content.

Format: D <starting address> <ending address>
D <starting address> <L number of bytes>

The starting and ending addresses may be given as offsets in the data segment. If access to another segment is required then the segment information should be included in the address, example: F CS:100 1FF 20

E – used to enter information in memory.

Format: E <address> <data list>
E <address>

If the E command is used without the data list, DEBUG assumes that you wish to examine that byte of memory and possibly modify it. The following options are given to you in that case:

a – You may enter a new data byte which DEBUG will write to memory.

b – You may press <ENTER> to signify you do not wish to modify the byte.

c – You may press the space bar which will leave the displayed byte unchanged and move to the next byte where you may possibly modify it.

d – You may enter the minus sign, which will leave the displayed byte unchanged and move you to the previous byte where you may possibly modify it.

See examples below:

```
-F 100 11F 20<ENTER>
-F 120 13F 30<ENTER>
-D 100 13F<ENTER>
0B2C:0100  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
0B2C:0110  20 20 20 20 20 20 20 20-20 20 20 20 20 20 20 20
0B2C:0120  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30
0000000000000000
0B2C:0130  30 30 30 30 30 30 30 30-30 30 30 30 30 30 30 30
0000000000000000
-F 140 L20 31<ENTER>
-D 140 L20<ENTER>
0B2C:0140  31 31 31 31 31 31 31 31-31 31 31 31 31 31 31 31
1111111111111111
0B2C:0150  31 31 31 31 31 31 31 31-31 31 31 31 31 31 31 31
1111111111111111

-E 100 'Gabi'<ENTER>
-D 100 10F<ENTER>
0B2C:0100  47 61 62 69 20 20 20 20-20 20 20 20 20 20 20 20   Gabi
```

```
-E 100<ENTER>
0B2C:0100  47.67<ENTER>
-D 100 10F<ENTER>
0B2C:0100  67 61 62 69 20 20 20 20-20 20 20 20 20 20 20 20   gabi
-E 100<ENTER>
0B2C:0100  67.<SPACE BAR> 61.41<SPACE BAR>62.42<SPACE
BAR>69.49<ENTER>
-D 100 10F<ENTER>
0B2C:0100  67 41 42 49 20 20 20 20-20 20 20 20 20 20 20 20   gABI
-
```

12 – Fill the memory locations 100 to 12F with the ASCII character which represents the number 5, then display the modified memory locations.
13 – Enter EE3751 in memory location 130, and then display those memory locations. Then using the E 130 command, modify the EE characters to ee. Notice that the data list was excluded. Demonstrate to the instructor the procedures performed above.

Using Redirection with DEBUG
14 – Use Notepad to create a file called INPUT.TXT with the following contents:
F 150 L10 41<ENTER>
F 160 L10 61<ENTER>
D 150 L20<ENTER>
Q<ENTER>

Make sure this file is in the C:\WORK subdirectory or in the A: drive, whichever one you are using. Issue the following command at the DOS prompt:

```
\A:\DEBUG < INPUT.TXT<ENTER>
-F 150 L10 41
-F 160 L10 61
-D 150 L20
0B92:0150  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0B92:0160  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61
aaaaaaaaaaaaaaaa
-Q
A:\
```

Issue the following command next:
```
A:\DEBUG < INPUT.TXT > OUTPUT.TXT<ENTER>
A:\
```

Open the file OUTPUT.TXT with Notepad and check its contents.

Print file and submit it to TA.

Editing, assembling, linking and executing an Assembly Language program.

```
                .MODEL SMALL
                .386
                .STACK 64
                .DATA
MESS            DB      'Hello World!',13,10,'$'
                .CODE
BEGIN           PROC FAR
                MOV AX,@DATA
                MOV DS,AX

                MOV AH,9H
                MOV DX,OFFSET MESS
                INT 21H

                MOV AH,4CH
                INT 21H
BEGIN           ENDP
                END BEGIN
```

1 – Copy the program above using **Notepad**, and save it as **PROG0A.ASM** in the **A:\>** drive.

2 – Following you will find the commands that you should type to accomplish the task of assembling, linking and using DEBUG to execute the program given to you.

```
A:\>TASM PROG0A.ASM /L<ENTER>
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International
Assembling file:   PROG0A.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  435k
```

3 - If there are any problems during assembly you will see several error messages displayed on a DOS window opened by the OS. At this point you may open the file **PROG0A.LST** to check where the errors occur and then edit them in the **PROG0A.ASM** file, before assembling it again, otherwise continue below.

```
A:\>TLINK PROG0A.OBJ<ENTER>
Turbo Link  Version 5.1  Copyright (c) 1992 Borland International
```

4 – Now you are going to use DEBUG to execute the program.

```
A:\>DEBUG PROG0A.EXE<ENTER>
-G<ENTER>
Hello World!

Program terminated normally
-Q<ENTER>
A:\>
```

5 – You can also execute the program directly by doing what is shown below:

```
A:\>PROG0A.EXE<ENTER>
Hello World!

A:\>
```

6 – Copy the program below using **Notepad**, and save it as **PROG0B.ASM** in the **A:\>** drive.

```
                .MODEL SMALL
                .386
                .STACK 64
                .DATA
N1              DB      33H
N2              DB      24H
SUM             DB      0H
                .CODE
BEGIN           PROC FAR
                MOV AX,@DATA
                MOV DS,AX

                MOV AL,N1
                ADD AL,N2
                MOV SUM,AL

                MOV AH,4CH
                INT 21H
BEGIN           ENDP
                END BEGIN
```

7 – Following you will find the commands that you should type to accomplish the task of assembling, linking and using DEBUG to execute the program given to you.

```
A:\>TASM PROG0B.ASM /L<ENTER>
Turbo Assembler  Version 3.1  Copyright (c) 1988, 1992 Borland
International
```

```
Assembling file:   PROG0B.ASM
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  435k
```

8 - If there are any problems during assembly you will see several error messages displayed on a DOS window opened by the OS. At this point you may open the file **PROG0B.LST** to check where the errors occur and then edit them in the **PROG0B.ASM** file, before assembling it again, otherwise continue below.

```
A:\>TLINK PROG0B.OBJ<ENTER>
Turbo Link  Version 5.1  Copyright (c) 1992 Borland International
```

9 – Now you are going to use DEBUG to execute the program and to verify memory to see if the program executed correctly. Follow the steps below exactly.

```
A:\>DEBUG PROG0B.EXE<ENTER>
```

10 – The next two steps allow you to find out the initial address of the data segment, and its contents. You will do this by disassembling the first instruction (MOV AX,@DATA) of your program, then dumping the contents of several memory locations starting at the address you just found.

```
-U CS:0 1<ENTER>
10A2:0000 B8A410          MOV AX,10A4
-D 10A4:0 2<ENTER>
10A4:0000  33 24 00                                              3$.
```

11 – To execute the program, and to display the memory, follow the steps below. These steps allow you to verify if the program executed correctly.

```
-G<ENTER>

Program terminated normally
-D 10A4:0 2<ENTER>
10A4:0000  33 24 57                                              3$W
```

12 – Quitting DEBUG.

```
-Q<ENTER>
A:\>
```

Procedure

Using the information learned from the previous steps, modify **PROG1B.ASM** to perform SUM = A + B – C.

DATA segment array definition.

```
.DATA
A       DW    1234H
B       DW    4321H
C       DW    1331H
SUM     DW    ?
```

Using DOS redirection and DEBUG, execute your program, then dump the contents of the data segment to a file called DUMP.TXT. Include the printout of this file with your program.

Experiment 1 – Subroutines, and DOS Interrupts

Subroutines (Procedures)

Subroutines are groups of instructions that usually perform one task. These tasks may be reused as often as necessary and may be called from anywhere in the program. Memory is saved by using procedures, but one disadvantage is the longer execution time needed for the computer to link to the procedure and to return to the location where the procedure was called. Two instructions are used with procedures: CALL, used to execute the procedure, and RET, used to return program execution to the point following the CALL statement.

Subroutines are defined as shown by the statements colored red:

```
DELAY PROC NEAR
        PUSH ECX
        MOV ECX,07FFFFH
AGAIN:
        LOOP AGAIN
        POP ECX
        RET
DELAY ENDP
```

The main procedure is identified by the last statement of the program, therefore the order or arrangement of all procedures within the program does not matter. To make grading easier lets use the arrangement displayed below, in other words main procedure on top followed by the subroutines.

```
                .MODEL SMALL
                .386
                .STACK 64
                .DATA
MESS            DB      'Hello World!',13,10,'$'
                .CODE
BEGIN           PROC FAR                    ;Main procedure
                MOV AX,@DATA
                MOV DS,AX

                MOV AH,9H
                MOV DX,OFFSET MESS
                INT 21H
                CALL DELAY

                MOV AH,4CH
                INT 21H
BEGIN           ENDP
```

```
DELAY PROC NEAR                    ;Subroutines
        PUSH ECX
        MOV ECX,07FFFFH
AGAIN:
        LOOP AGAIN
        POP ECX
        RET
DELAY ENDP

        END BEGIN
```

DOS INT 21H

INT 21H is provided by DOS, and it is stored in DRAM when the operating system is loaded. The user can invoke this interrupt to perform several useful functions, like inputting data from keyboard, and outputting data to monitor. These interrupt subroutines may be used by issuing a software interrupt call (**INT *type***). The user will have to identify which function is being used by setting the AH register to a specific value. Other registers may also need to be modified. Following are the descriptions of several of the most common INT 21H functions.

INT 21H Function 01H: Inputting a single character from the keyboard with echo.

AH = 01H
AL = inputted ASCII character code

Code example to input a single character from keyboard and to echo it to the display.

```
MOV AH,01H
INT 21H
```

INT 21H Function 02H: Outputting a single character to the monitor.

AH = 02H
DL = ASCII character code to be displayed

INT 21H Function 09H: Outputting a string terminated with $ to the monitor.

AH = 09H
DX = String address

INT 21H Function 4CH: Terminate a process (EXIT).

AH = 4CH
AL = binary return code

This interrupt terminates a process and returns control to DOS or the parent process.

Procedure

Write a program to generate the first ten numbers of the alternating series given below, store them in an array called ALTERNATE, and display them to the PC monitor.

$$A(n) = \sum_{n=1}^{10} (-1)^{n+1} n^2$$

To display the computed Alternate series of numbers, we need to convert them to decimal by dividing consecutively by ten until achieving a zero quotient, see below an example showing the procedure using the number 75:

	Quotient	Remainder
1001011/1010	111	101
111/1010	0	111

Store the results in an array called Decimal. Note that each number will require two bytes for storage.

Finally convert the decimal numbers to ASCII, store them in the ASCII array and display them to the PC monitor. All the numbers should be displayed one per line as shown below:

Alternate(1) = 1
.
.
Alternate(10) = -55

The program should have 5 subroutines:

GEN_ALT: to generate Alternate(X) where: $1 \leq X \leq 10$;
- Use register AH to pass the X value to the subroutine;
- Use register AL to return Alternate(X) to calling routine;

CONV_DEC: to convert an 8 bit binary number valued between 00000000B and 01100011B to a two digits decimal number represented by two 8 bits binary numbers valued between 00000000B and 00001001B;
- Use register AL to pass number to be converted to decimal to subroutine;

- Use registers AH to return MSD(most significant digit) and AL to return LSD(least significant digit);

CONV_ASCII: to convert a BCD number to ASCII;

- Use register AL to pass number to subroutine and AH to return ASCII value;

DISPLAYONE: to display a single character to screen.

- Use register DL to pass ASCII value to subroutine.

DISPLAYALL: to display a string of characters terminated by a $ to the screen.

- Use register DX to pass the address of the string of characters to the subroutine.

Array definition:

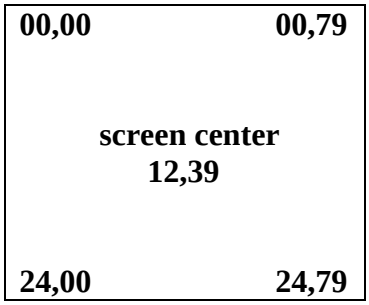
```
ALTERNATE       DB      0,1,8 DUP(?)
DECIMAL         DW      10 DUP(?)
ASCII           DW      10 DUP(?)
```

Experiment 3 – BIOS Interrupts.

Included within the BIOS one will find a set of interrupts that are extremely useful. These interrupt subroutines may be used by issuing a software interrupt call (**INT *type***). The user will also have to identify which function is being used by setting the AH register to a specific value. Other registers may also need to be modified. One should note that interrupts are a FAR call, in other words, when an interrupt is executed the return address is stored as CS:IP.

BIOS INT 10H

INT 10H are stored in the BIOS ROM of the IBM PC type computers, and they are used to control the screen video. The monitor screen in normal text mode is composed of 25 rows and 80 columns, and text mode is the default mode whenever a monitor is turned on. There are several types of monitors including: MDA, MCGA, CGA, EGA and VGA. In all these modes the text screen is 80X25 characters long. The text locations are numbered from 0 to 24 for the rows and 0 to 79 for the columns as shown in the diagram below.



Several functions are performed by INT 10H, therefore the programmer needs to identify which one is being used by storing an appropriate value in register AH. For example:

```
AH = 00H      ;Selects the change video mode function
INT 10H       ;Executes BIOS interrupt 10H.
```

Depending on the function being used, other register may be used to pass information to the interrupt subroutine. Following are the descriptions of several of the most common INT 10H functions.

INT 10H Function 00H: Change video mode

AH = 00H
AL = Video Mode
- 03H – 80X25 CGA text
- 07H – 80X25 Monochrome text.

Code example to set video mode to 80X25 CGA text

```
MOV AL,03H
MOV AH,00H
INT 10H
```

INT 10H Function 02H: Set Cursor Position

AH = 02H
BH = Page number
DH = Row number
DL = Column number

BH = 0 when graphics mode is in use.

INT 10H Function 06H: Scroll window up
INT 10H Function 07H: Scroll window down

AH = 06H to scroll up or 07H to scroll down
AL = Number of lines to scroll
BH = Display attribute
CH = Y coordinate of top left
CL = X coordinate of top left
DH = Y coordinate of lower right
DL = X coordinate of lower right

If AL = 0 the entire window is blank, otherwise, the screen will be scrolled upward/downward by the value stored in AL. Lines scrolling off the screen are lost and blank lines are scrolled in at the bottom/top according to the attribute in BH.

INT 10H Function 08H: Read character and attribute at cursor position

AH = 08H
BH = Display page

AH = Returned attribute byte
AL = Returned ASCII character code

INT 10H Function 09H: Write character and attribute at cursor position

AH = 09H
AL = ASCII character code
BH = Display page
BL = Attribute
CX = Number of characters to write

The character attribute is defined as shown in the following tables:

Monochrome display attributes

Blinking	Background			Intensity	Foreground		
D7	D6	D5	D4	D3	D2	D1	D0

D7 Non-blinking= 0
Blinking = 1
D3 Normal intensity = 0
Highlighted intensity = 1
D6 D5 D4 and D2 D1 D0 White = 0 0 0
Black = 1 1 1

CGA display attributes

Blinking	Background			Intensity	Foreground		
	R	G	B		R	G	B
D7	D6	D5	D4	D3	D2	D1	D0

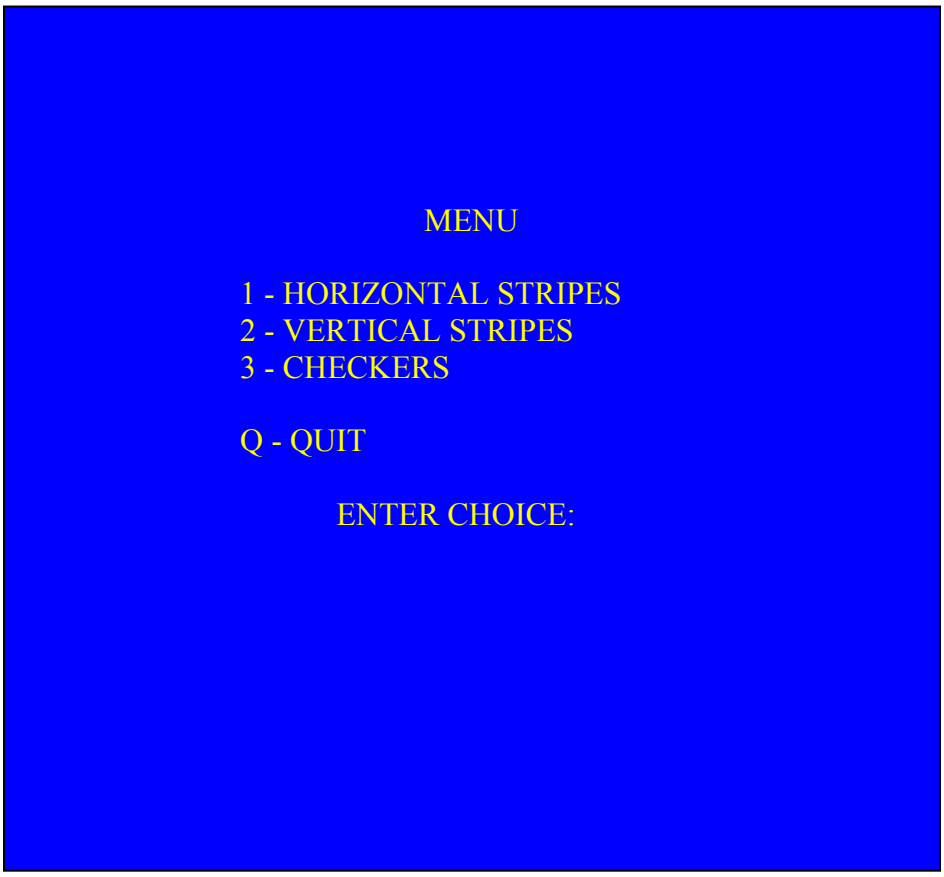
D7 Non-blinking= 0
Blinking = 1
D3 Normal intensity = 0
Highlighted intensity = 1
Both blinking and intensity are applied to foreground only.
D6 D5 D4 and D2 D1 D0 Color as defined on the following table

Color Attributes

I	R	G	B	Color
0	0	0	0	Black
0	0	0	1	Blue
0	0	1	0	Green
0	0	1	1	Cyan
0	1	0	0	Red
0	1	0	1	Magenta
0	1	1	0	Brown
0	1	1	1	White
1	0	0	0	Gray
1	0	0	1	Light blue
1	0	1	0	Light green
1	0	1	1	Light cyan
1	1	0	0	Light red
1	1	0	1	Light magenta
1	1	1	0	Yellow
1	1	1	1	High intensity white

Procedure

Using the interrupts described above, write a program to:
1 - Clear the screen.
2 – Set the video mode to 80X25 CGA text.
3 - Create the following menu of choices:



The background color is blue and the foreground color for the letters is yellow.

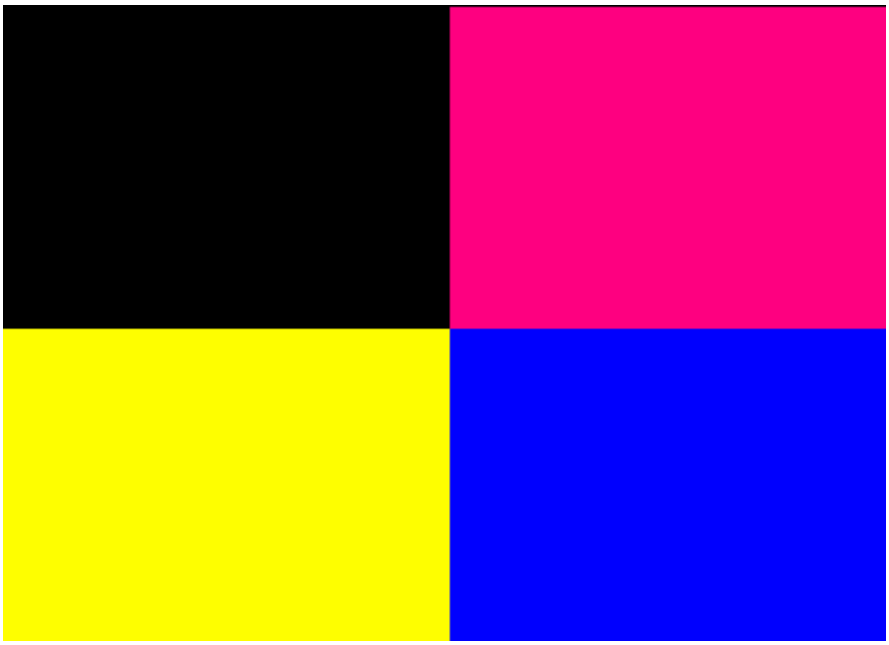
4 – Change the screen according to the choice selected. Next page contains a couple of examples. Display the new screen until any key is pressed on the keyboard then return to the main screen to display the menu of choices again.

Several subroutines will be created for this program, and they are described below:

CLEAR_SCREEN – this subroutine will clear the screen;
CHG_ SCREEN_ATTRIB – this subroutine will change the attribute of a rectangular area of the screen. Use the following registers to pass arguments to subroutine:
BL – attribute;
CX – upper left coordinate (CH – row, CL- column);
DX – lower right coordinate (DH – row, DL- column);

DISP_MESS – this subroutine display messages to the screen. Use register DX to pass the initial address of the message to displayed to the subroutine.

Choose colors that appeal to you. In other words, you do not have to use the same colors of the examples if you do not want to.



Experiment 4 – Introduction to String Instructions

In this experiment, software to manipulate data entered into a buffer will be developed using string instructions. A couple subroutines will be developed to simulate the DOS function keys F1, F3 and F5.

The 386 instruction set contain a group of instructions to perform string manipulation with auto increment/decrement addressing. Two dedicated registers, DI and SI, are used for this purpose, and they are used as pointers to the data being manipulated. Note that when data is to be stored in memory the DI (Destination Index) register points, relative to the ES register, to the memory location. If data is to be read from memory, then the SI (Source Index) register points, relative to the DS register to the memory location. Some instructions may use both DI and SI registers as pointers. These register are automatically incremented or decremented depending on the value of the direction flag If the direction flag is set (DF = 1) then the registers are decremented, if the direction flag is clear (DF = 0) then the registers are incremented. The direction flag may be set with the STD instruction and cleared with the CLD instruction.

Below you will find a brief description of some string instructions:

LODSB/LODSW/LODSD/LODS – Load string byte/word/double word.
This instruction loads a value from memory into the AL, AX or EAX register. If the direction flag is clear the SI register will be incremented by 1, 2, or 4 depending on whether the instruction is operating on a byte, word or double word respectively. Otherwise, if the direction flag is set the SI register will be decremented by 1, 2, or 4 again with respect of the size of data being manipulated.

AL/AX/EAX=DS:[SI];SI±1/2/4

STOSB/STOSW/STOSD/STOS – Store string byte/word/double word.
This instruction stores a value from the AL, AX or EAX register into memory. If the direction flag is clear the DI register will be incremented by 1, 2, or 4 depending on whether the instruction is operating on a byte, word or double word respectively. Otherwise, if the direction flag is set the DI register will be decremented by 1, 2, or 4 again with respect of the size of data being manipulated.

ES:[DI]=AL/AX/EAX;DI±1/2/4

SCASB/SCASW/SCASD/SCAS – Compares a string byte/word/double word in memory to the contents of the AL/AX/EAX registers and modifies the flags accordingly. Note that the contents of the memory and the AL/AX/EAX register are not modified, and that the memory location is pointed to by the DI register relative to the ES register. Also the DI register is automatically incremented/decremented dependent on the value of the direction flag DF=0/1 respectively. The two operands are actually compared by subtracting the value of the memory location pointed to by DI from the value stored in the AL/AX/EAX register.

REP/REPZ/REPNZ - These prefixes cause the string instruction that follows them to be repeated the number of times in the count register ECX or until:

ZF=0 in the case of REPZ (repeat while equal).
ZF=1 in the case of REPNZ (repeat while not equal)**.**

The following code segment searches a buffer for the character ’f’:

```
MOV AL,’f’
LEA DI,BUFFER
MOV ECX,6
CLD
REPNE SCASB
```

Procedure

Write 4 subroutines, described below, and some code segments to complete the given code:

FILL_BUFFER – This subroutine will fill the buffer with characters entered from the keyboard. The maximum number of characters in the buffer is 80. The carriage return control code is to be used to identify that the user has finished typing the string to be stored in the buffer and that control of the program should be returned to the calling routine.
DISP_ONE – This subroutine will copy a single character from the buffer to the screen, then return program control to main routine. The character to be displayed is pointed to by the buffer pointer, which should be updated after the character is copied to the screen.
DISP_ALL – This subroutine will display the contents of the buffer from the current location of the buffer pointer to the end of the buffer, then update buffer pointer to point to the beginning of the buffer and return control of program to the main routine.
SEARCH_ONE – This subroutine will search the buffer for a character to be inputted from the keyboard by the user. If the character is found the buffer pointer is updated to point to it and control of the program should be returned to main routine. If the character is not found display mess3 and return control of program to main routine.

Use the screen diagram shown below. The yellow area is used to display the characters entered in the buffer and to display the functionality of each subroutine.

The code given below is incomplete, so modify it at will.

```
.MODEL SMALL
.386
.STACK 64
.DATA
NL              DB      10,13,’$’
```

```
MENU        DB    'MENU',10,13
            DB    'Press I start filling buffer.',10,13
            DB    '     1 to display a single character',10,13
            DB    '     3 to display remainder of buffer',10,13
            DB    '     5 to search for a single character',10,13,10,13
            DB    '     Q to quit program.',10,13, ,10,13,
            DB    '     ENTER CHOICE: ',10,13,'$
MESS        DB    'Character not found.',10,13,'$'
BUFFER      DB    80,?,80 DUP(0FFH)
.CODE
BEGIN PROC FAR
            MOV AX,@DATA
            MOV DS,AX

;Routine to modify the screen attribute



AGAIN:      CALL CLEAR_SCREEN
            MOV DH,3
            MOV DL,0
            MOV AH,2
            INT 10H
            MOV DX,OFFSET MENU
            CALL MESSAGE
            CALL GET_CHAR
            CMP AL,'I'
            JNE AGAIN

            MOV DH,13
            MOV DL,0
            MOV AH,2
            INT 10H

            CALL FILL_BUFFER

NEXT:       MOV DH,3
            MOV DL,0
            MOV AH,2
            INT 10H
            MOV DX,OFFSET MENU
            CALL MESSAGE

            CALL GET_CHAR
            CMP AL,'I'
```

```
                JNE C1
                CALL FILL_BUFFER
                JMP NEXT
C1:             CMP AL,'1'
                JNE C3
                CALL DISP_ONE
                JMP NEXT
C3:             CMP AL,'3'
                JNE C5
                CALL DISP_ALL
                JMP NEXT
C5:             CMP AL,'5'
                JNE CQ
                CALL SEARCH_ONE
                JMP NEXT
CQ:             CMP AL,'Q'
                JNE NEXT

;Exit program
                MOV AH,4CH
                INT 21H
BEGIN ENDP

;Fill buffer subroutine



;Display one character subroutine



;Display buffer subroutine



;Search for a character subroutine
```

```
;Input from keyboard subroutine
;               inputs a single character from keyboard and display character to monitor
;               character read is returned to calling routine in AL register
GET_CHAR PROC NEAR
                MOV AH,1
                INT 21H
                RET
GET_CHAR ENDP

;Clear screen subroutine
CLEAR_SCREEN PROC NEAR
                PUSH AX
                PUSH BX
                PUSH CX
                PUSH DX
                MOV AH,6
                MOV AL,0
                MOV BH,7
                MOV CH,0
                MOV CL,0
                MOV DH,24
                MOV DL,79
                INT 10H
                POP DX
                POP CX
                POP BX
                POP AX
                RET
CLEAR_SCREEN ENDP

;Message subroutine
;               sends a message to the CRT display
;               pointer to message should be in DX register
MESSAGE PROC NEAR
                PUSH AX
                CALL NEW_LINE
                MOV AH,9
                INT 21H
                POP AX
                RET
;MESSAGE ENDP

;New line subroutine
;               moves the CRT display's cursor to the beginning of a new line
NEW_LINE PROC NEAR
```

```
        PUSH AX
        PUSH DX
        MOV DX,OFFSET NL
        MOV AH,9
        INT 21H
        POP DX
        POP AX
        RET
NEW_LINE ENDP

END BEGIN
```

Experiment 5 – String Instructions

In this lab you will learn to use a few more string instruction, which are described below:

MOVSB/MOVSW/MOVSD/MOVS – Moves a string byte/word/double word located at an address pointed to by the SI register relative to the DS register, to another address pointed to by the DI register relative to the ES register. If the direction flag is clear the SI and DI registers will be incremented by 1, 2, or 4 depending on whether the instruction is operating on a byte, word or double word respectively. Otherwise, if the direction flag is set the SI and DI registers will be decremented by 1, 2, or 4 again with respect of the size of data being manipulated.

ES:[DI]=DS:[SI];SI±1/2/4;DI±1/2/4

CMPSB/CMPSW/CMPSD/CMPS – Compares a string byte/word/double word located at an address pointed to by the SI register relative to the DS register, to another string byte/word/double word located at an address pointed to by the DI register relative to the ES register and modifies the flags accordingly. Note that the contents of the memory are not modified, and that the SI and DI registers are automatically incremented/decremented dependent on the value of the direction flag DF=0/1 respectively. The two operands are actually compared by subtracting the value of the memory location pointed to by DI from the memory location pointed to by SI.

Procedure

Modify the program written in experiment 4 in the following way:

1 – Create a menu with the following choices:

MENU

I – Enter data in the buffer.

1 – Display a single character from the buffer.

3 – Display the whole buffer.

5 – Search buffer for a character entered by the user.

7 – Compare string

9 – Replace string.

Q – Quit program.

Enter choice:

A - The screen set up should be similar to the one of the previous experiment. The work area is the yellow one and all data entered should be displayed in this area. Notice that the menu and work areas have defined locations that cannot be changed during the execution of the program. You may write new data over information already displayed or erase the information by writing spaces over them. Use interrupt 10H to set cursor position where you need to write or modify what is displayed.

B – Validate choice entered. If a correct choice was entered perform the action required, and display the results in the work area. Return to the menu to receive another selection. If a wrong choice was made, display the message WRONG CHOICE, TRY AGAIN in red letters, within the menu half of the screen. Reset the menu to its original form by erasing the error message.

C – Any information being entered by the user should be displayed in the work area. If you want you may select a specific line of the work area for that purpose. I will leave this choice up to you.

D – The result for options 7 should be a message indicating that the strings are equal or not equal.

E – For option 9 two strings should be entered by the user: a - string to be searched for, and b – a string to replace the one searched. Both strings should be displayed in the work area in different lines and the buffer with the replaced string should be displayed below both strings. Display appropriate messages in case strings are not found.

F – Make sure the screen does not scroll up or down. Note that if you write data to the bottom row of the screen and the data size exceeds the row length, DOS will wrap-around and scroll 1 line upwards so that the data will be completely displayed.

Observation: Make use of the DOS and BIOS interrupts to manipulate the display.

You are not limited to use just string instructions, but some of them will be much more useful to use.

Experiment 6 – Parallel Communication

Communication is required between a microprocessor and some device external to the computing system. This communication may be done serially or in parallel. In a parallel port, several signal lines are used to exchange information with other devices. This information is exchanged in multiple bits at a time, unlike a serial port where information is transferred one bit at a time over a single transmission line.

The parallel port provided with a PC was originally designed as a port to be interfaced to a printer. Nowadays, this port is being used to interface the PC to devices such as: scanners, video capture devices, external disk drives, test equipment (scopes, multimeters, etc), and other equipment.

Since its inception, the parallel printer port has being modified several times to make it a faster and more versatile means of interfacing devices to the PC. Today one may find the following types of parallel printer ports:

Original (SPP - Standard Parallel Port)
PS/2-type (simple bi-directional)
EPP – Enhanced Parallel Port
ECP – Extended Capabilities Port.

This experiment uses the SPP type port, which was the original parallel printer port, and it had the following layout. Three consecutive addresses defined as follows were used. The first address, the port *base address*, which is also called the Data register, is used to output information to 8 data lines. The second address, *base address* + 1, also called the Status register, is used to input the state of 5 status lines. The third address, *base address* + 2, also called the Control register, is used to output values into 4 control lines. The standard *base addresses* are: 278h, 378h and 3BCh. Table 1 in Appendix B describes the different signals, arranged by registers, which are used by the SPP.

One may access ports with the Intel 80x86 family of microprocessors, by the use of the **IN** and **OUT** instructions. Their formats are given below:

IN Destination, Source

Where:

Source can be an immediate address or the address in the DX register.

Destination can be the data in the AL, AX, or EAX registers.

Example: MOV DX,379H

IN AL,DX

Reads the value in port 379H into the AL register.

OUT Destination, Source

Where:

Source can be the data in the AL, AX, or EAX registers.

Destination can be an immediate address or the address in the DX register.

Example: MOV DX, 378H

MOV AL,'A'

OUT DX,AL

Transfers the letter "A" to the port in address 378H.

Appendix B contains more information about the parallel port.

Procedure

Write a program to interface to a circuit connected to the parallel port. The numbers outputted to the parallel port will be displayed in eight LEDs connected to the data register. A lit LED signifies a logic 1 value, while an unlit one signifies a logic 0.

Assemble the circuit shown in the logic diagram below. Make sure your circuit's ground and the computer ground are the same.

Your program should display the menu choices shown in the screen diagram below. Following is an explanation of each menu choice:

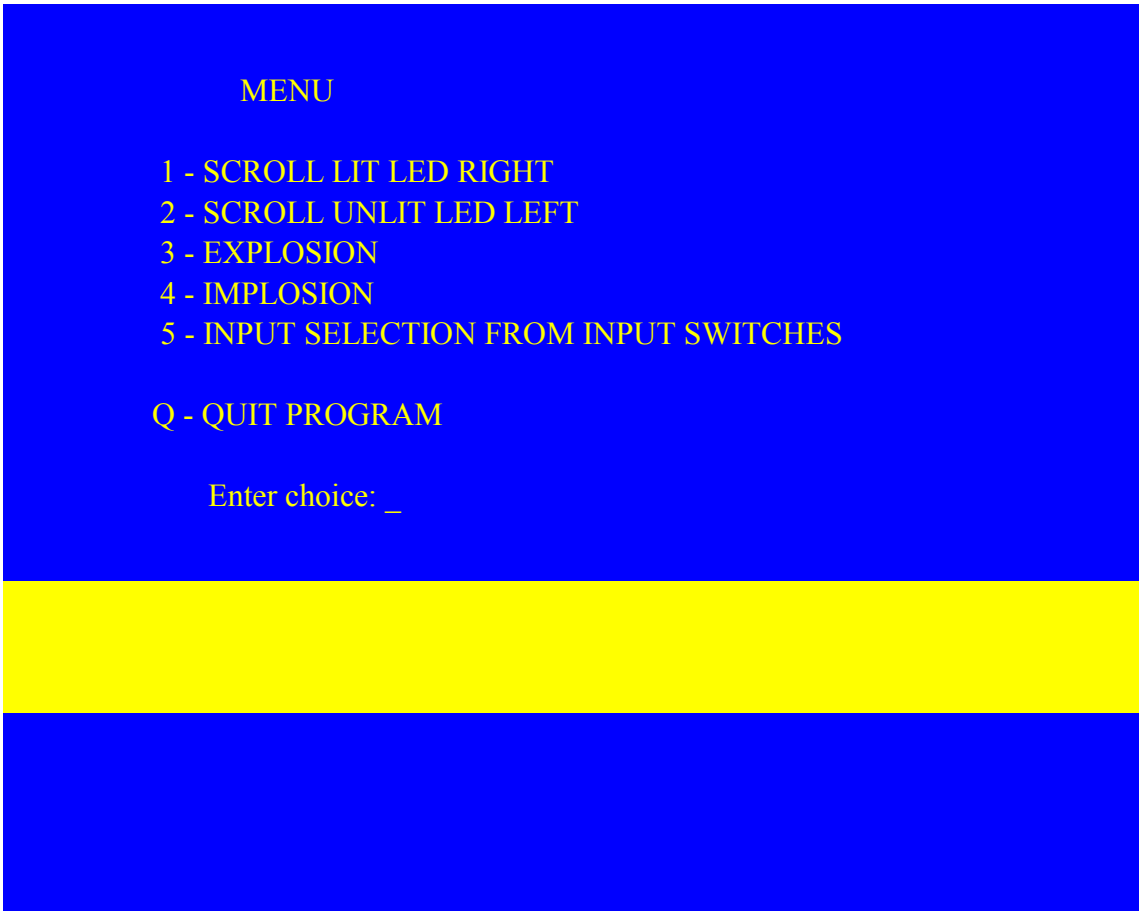
1 – SCROLL LIT LED RIGHT – the rightmost led will be turned on then off. This action will be repeated with the next rightmost led, then the next, until the leftmost led is reached.

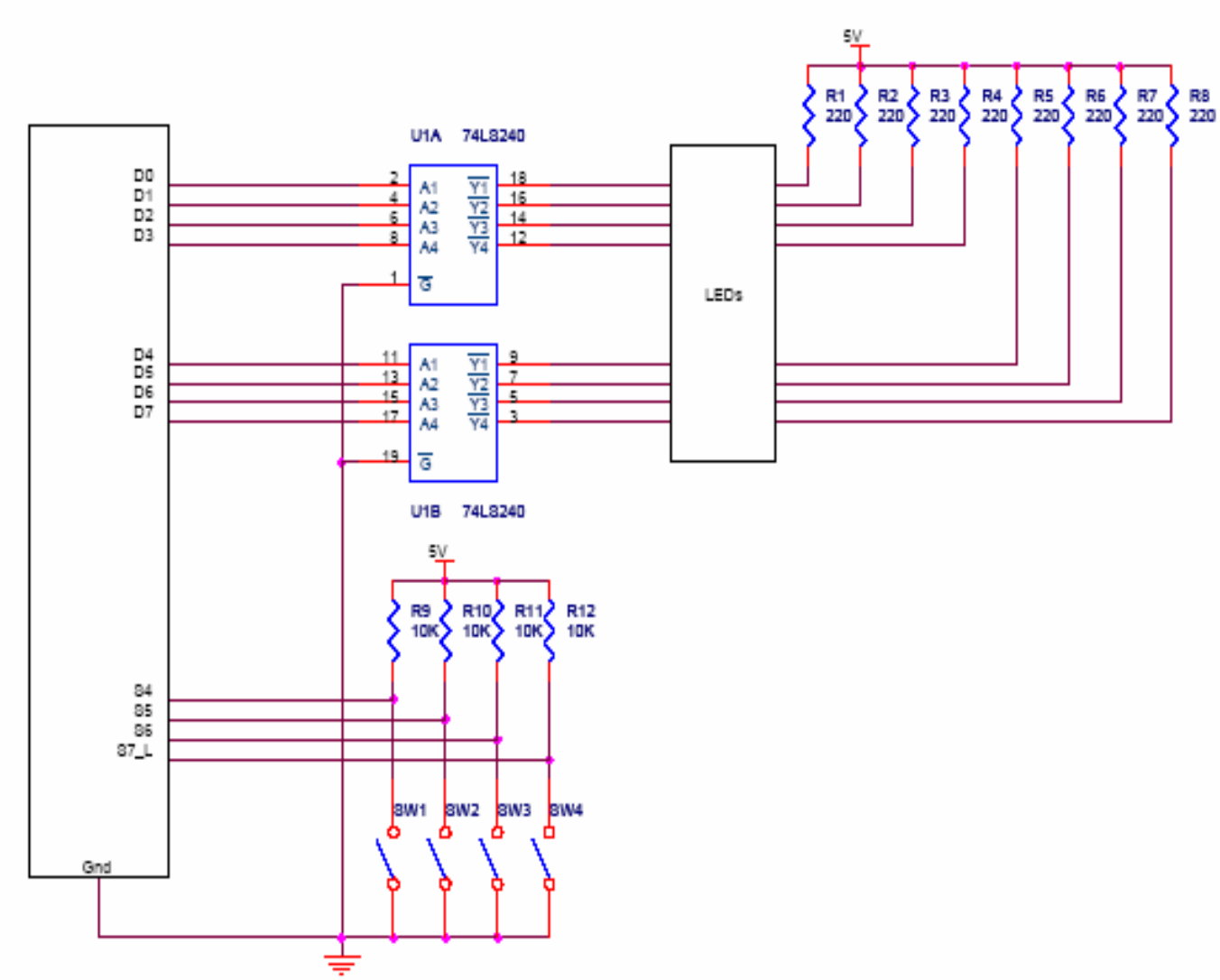
2 – SCROLL UNLIT LED LEFT – turn all leds on, then turn the leftmost led off then on. This action will be repeated with the next leftmost led, then the next, until the rightmost led is reached.

3 – EXPLOSION – start by turning the centermost leds on. Turn them off then on a couple of times. Continue the procedure by repeating the action performed on the two centermost leds with the 4 centermost, then the 6 centermost, until all the leds in the bar are used.

4 – IMPLOSION – reverse the explosion procedure.

5 – INPUT SELECTION FROM CIRCUIT SWITCHES – output “Set the switches, then press any key.” Message to the screen. Input the value set on the switches and use it to identify which selection from 1 to 4 is to be performed.




5V
R1 220
R2 220
R3 220
R4 220
R5 220
R6 220
R7 220
R8 220
U1A 74LS240
D0
D1
D2
D3
D4
D5
D6
D7
LEDs
U1B 74LS240
5V
R9 10K
R10 10K
R11 10K
R12 10K
S4
S5
S6
S7_L
SW1
SW2
SW3
SW4
Gnd

Experiment 7 – Parallel Communication

The PS2 protocol modified the parallel port in a very important way. It allowed the data register to become bidirectional thus allowing it to output data to an external device such as some LEDs or to input values from a keyboard. This modification entailed the addition of some extra bits in the control register with the sole purpose of enabling interrupt (bit 4) and making the port an input port bit 5). The table shown on the next page describes what the new bits mean. Notice that this bits are used internally by the computer parallel port circuitry and they are not accessible from the outside.

Procedure

Write a program to do the following:

The Main routine will set up a menu with the following choices:

MENU

A – TEST CIRCUIT.

Q – QUIT.

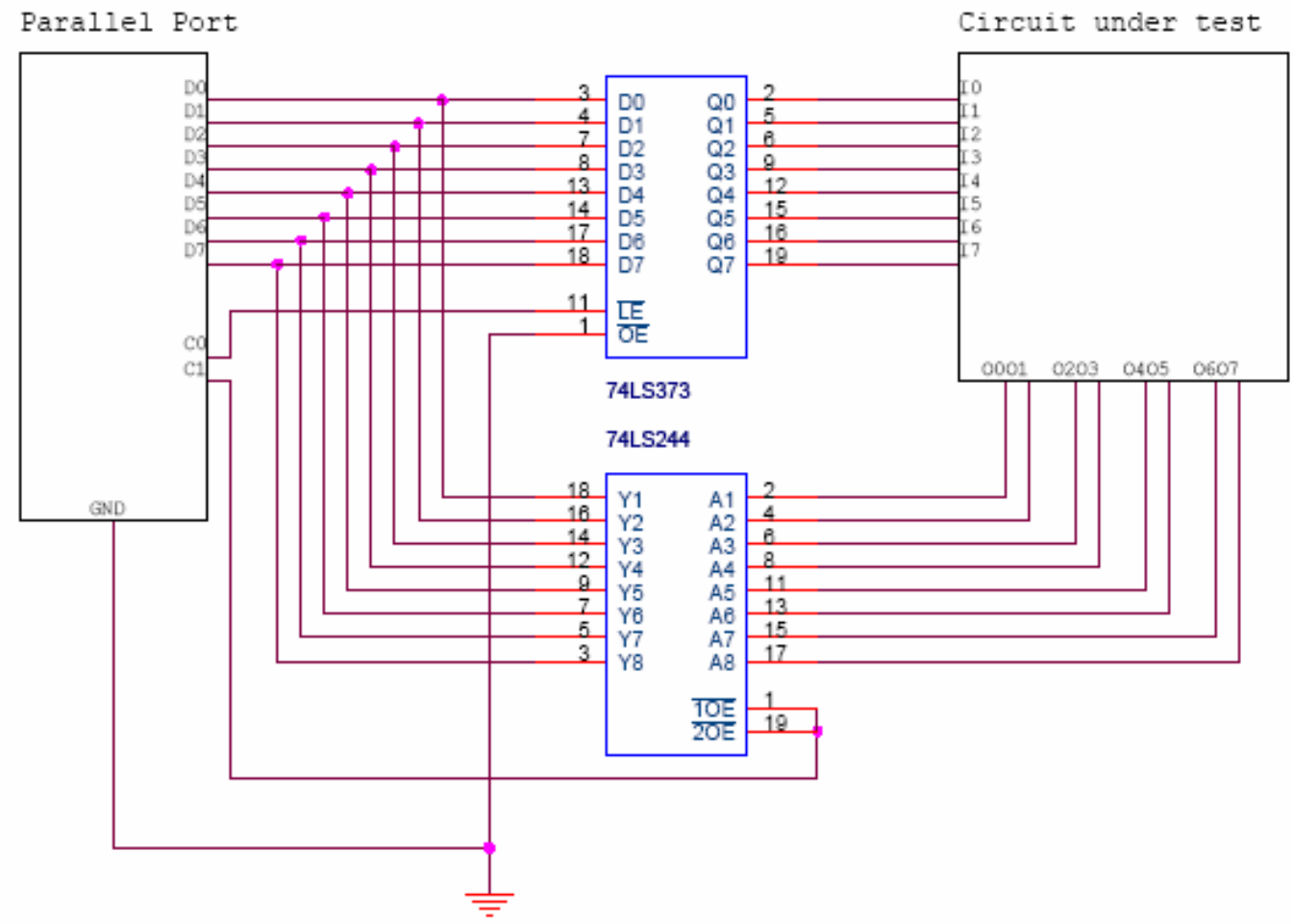
ENTER CHOICE:

Explanation of choices:

TEST CIRCUIT executes a subroutine used to test if the circuit is working properly. It should display a message stating the result of the test.

QUIT stops program and return control to O.S.

Circuit diagram:



Circuit under test:

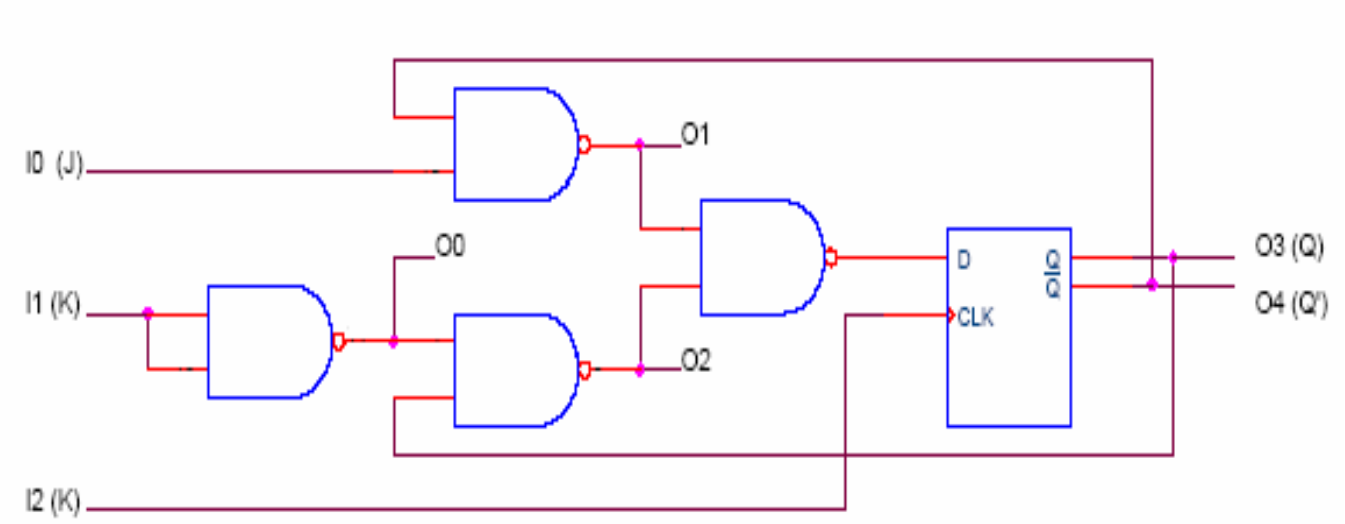


Figure 1

Data Register (Base Address)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	2	Data bit 0	No	Output
1	3	Data bit 1	No	Output
2	4	Data bit 2	No	Output
3	5	Data bit 3	No	Output
4	6	Data bit 4	No	Output
5	7	Data bit 5	No	Output
6	8	Data bit 6	No	Output
7	9	Data bit 7	No	Output
Status Register (Base Address + 1)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
3	15	nError	No	Input
4	13	Select	No	Input
5	12	PaperEnd	No	Input
6	10	nAck	No	Input
7	11	Busy	Yes	Input
Control Register (Base Address + 2)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	1	Nstrobe	Yes	Output
1	14	nAutoLF	Yes	Output
2	16	Ninit	No	Output
3	17	nSelectIn	Yes	Output
4		IRQ 1 = enabled		
5		Bidirectional 1 = input		
Ground Connections				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
	18 – 25	Ground		

Experiment 8 – Keyboard Interfacing

In this experiment, a mechanical-key switch keyboard is interfaced to the parallel port of a PC. Mechanical switches are relatively inexpensive, but they suffer several disadvantages: they are prone to contact bounce, which means that a pressed key will make and break contact several times before finally making solid contact; the contacts may oxidize or get dirty with age, so they no longer make a dependable connection.

Most keyboards organize the key switches in a matrix of rows and columns. Getting meaningful data from a keyboard such as this requires performing three major tasks:

1. Detection of a key press.

2. Debouncing of a key press.

3. Encoding the key press.

These three tasks can be done by software, hardware or a combination of both. In this experiment you will use the software method.

The circuit diagram shows the connections to be made to the parallel port of the PC. When no keys are pressed, the column lines are held high by pull-up resistors. The main principle here is that pressing a key connects a row to a column. If a low is output to a row and a key on that row is pressed then the low will appear on the column which contains the key, and this low can be detected on the input port. If you know the row and column of the pressed key, then you know which key was pressed, therefore you can encode the key with any value you wish.

Below is a description of what your program is supposed to do:

1. Output zeros to all of the rows.

2. Read the columns. If all columns are high go to 3 else repeat 2. This is done to ensure that a previously pressed key was released before looking for the next one.

3. Read the columns. If any column is low go to 4 else repeat 3.

4. Wait 20ms then read columns again. If the same value read in step 3 is found go to 5, else go to 2. This waiting period allows you to debounce the key.

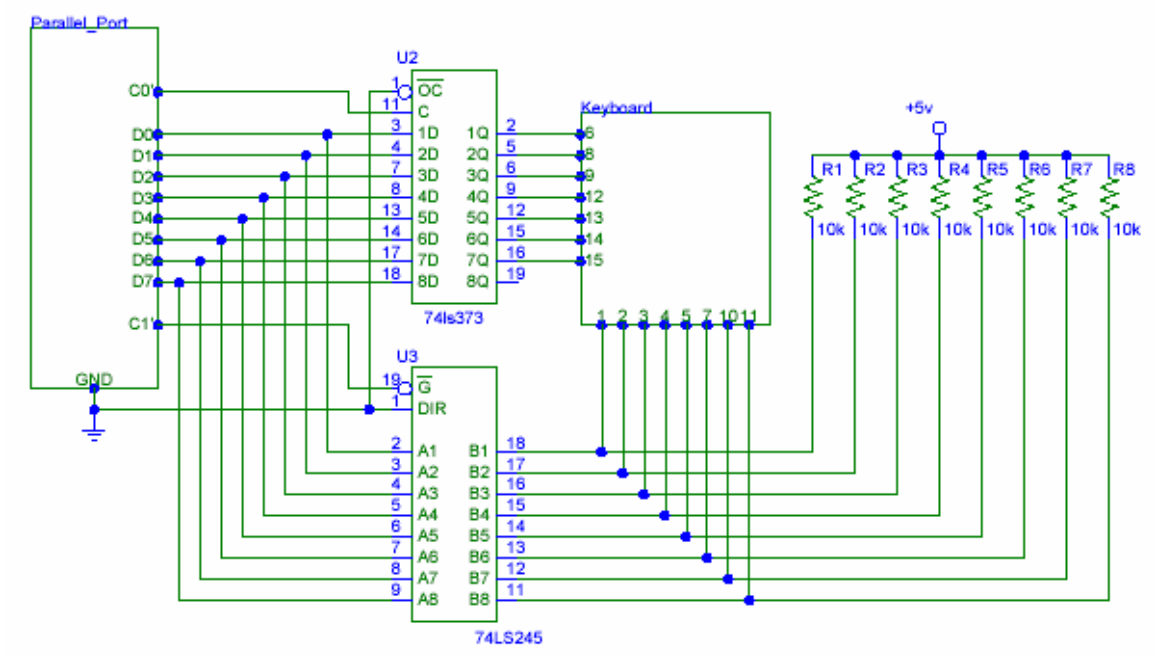
5. Identify which column contains the zero. This tells you which column the key is connected to.

6. To identify which row is connected to the key, output a zero to a single row then check if the column identified above becomes low, if it does then you found the desired row, else repeat 6 for the next row. Do this until all rows have been checked.

7. Now that both row and column have been identified, the key can be encoded.

Procedure

Assemble and test the circuit given in the schematic shown below, then write a program to detect, debounce, encode, and display a key pressed to the PC's monitor. Use the table below to encode the keys.



Keyboard Pin #	1	2	3	4	5	7	10	11
6						ALPHA LOCK		
8	P	0	A	;	/	\	Q	Z
9	Y	6	G	H	N	5	T	B
12	ENTER		SHIFT	SPACE	=	FCTN	CTRL	
13	O	9	S	L	.	2	W	X
14	!	8	D	K	,	3	E	C
15	U	7	F	J	M	4	R	V

Experiment 9 – Serial Communication

Computers use two methods to transfer information: serial and parallel. The parallel method allows the user to transfer information at faster rates but its hardware is more complex and expensive. Another problem with parallel communication is that the distance between the devices cannot be great. Serial communication overcomes some of the problem with parallel communication, so it is capable of transferring information at great distances and its hardware is less complex and cheaper. Serial communication can use the telephone network to send and receive information at great distances, due to the fact that it uses a single channel for information transfer. When using the telephone network the 0s and 1s that comprise the information being transferred have to be converted to audio tones. This function is performed by a device called modem, which stands for modulator/demodulator.

Two methods are used in serial communication:

Synchronous – information is transferred in block of data (characters) at a time;

Asynchronous – information is transferred one character at a time.

Special Ics have been designed by different manufactures to take care of serial communication. They are called:

UART –Universal asynchronous receiver-transmitter;

USART – Universal synchronous-asynchronous receiver-transmitter.

The transmission of data can be done in different ways. Simplex transmissions occur when a transmitter sends information to a receiver. The information flows in one way only, from the transmitter to the receiver. Information can also be transmitted and received simultaneously, over two transmission lines and this characterizes full duplex transmissions. Half duplex transmission is the method that uses a single transmission line to transfer data bi-directionally. Unlike the simplex method, the half duplex method allows information to flow both ways, one way at a time as shown in Figure -1.

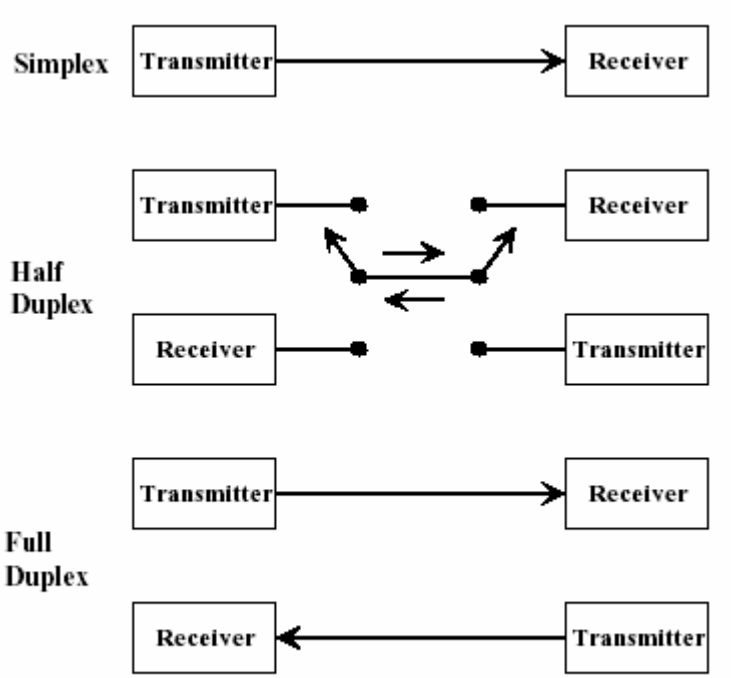


Figure - 1

In this experiment you are going to be using asynchronous serial communication. This form of serial communication is widely used for character-oriented transmissions. Each character is placed between a start and one or more stop bits, and this procedure is referred to as framing. The start bit is always a 0 (low) and the stop bits are always 1 (high). Between the start and stop bits one may find from 5 to 8 data bits and a parity bit in case parity is used for error checking.

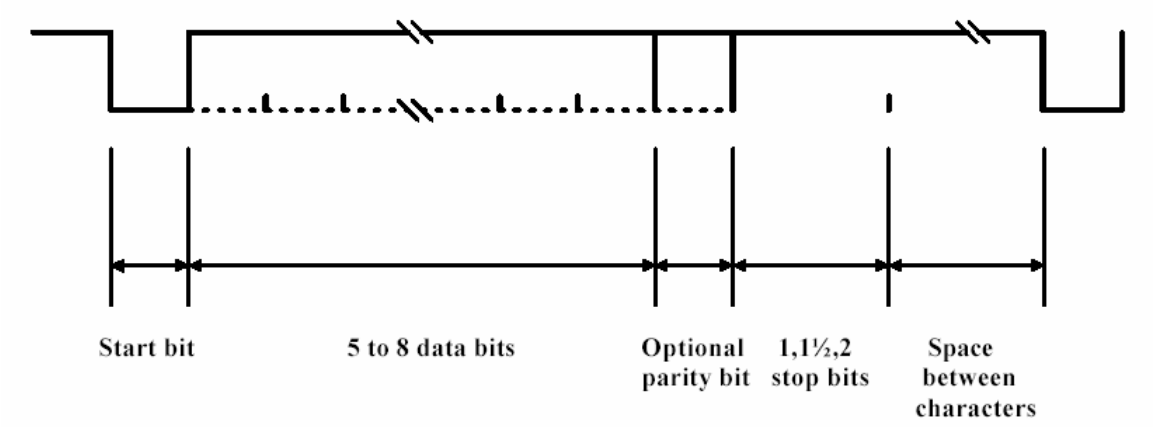


Figure - 2

As seen in the diagram shown above the same information may be exchanged with more or less bits in each frame. This requires that the devices being used in the communication

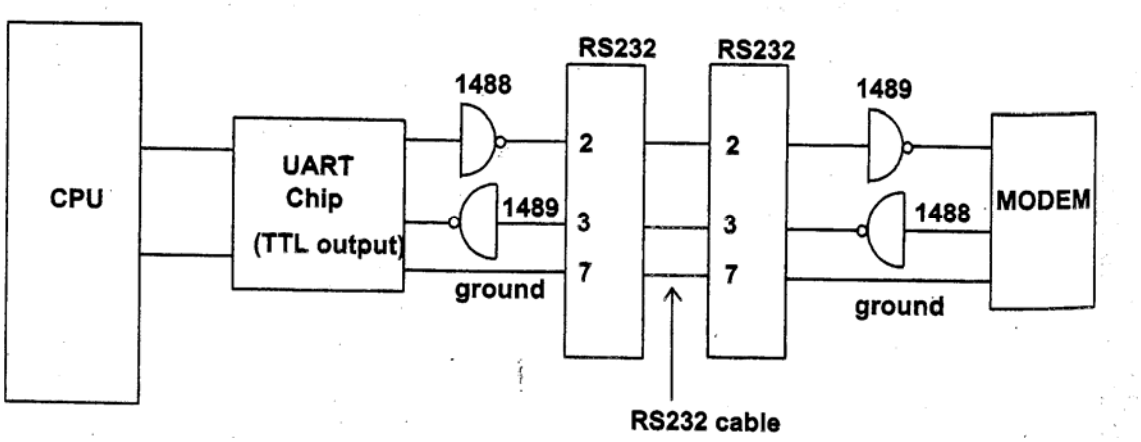
exchange be initialized in the manner the information will be transmitted and received. Four parameters need to be set:

1. Number of data bits used – 5, 6, 7, or 8.
2. Parity – even, odd or no parity.
3. Stop bits – 1, 1.5 or 2.
4. Baud rate – 110, 150, 300, 600, 1200, 2400, 4800 or 9600.

A computer transfers information using voltage levels that are TTL compatible, but serial communication may use many different standards for information transfer. The standard you will be exposed to is called RS232, and it represents information in the following way:

a. Logic 1 – voltages in the range –3V to –25V.
b. Logic 0 – voltages in the range 3V to 25V.
c. Voltage range between –3V and 3V is undefined.

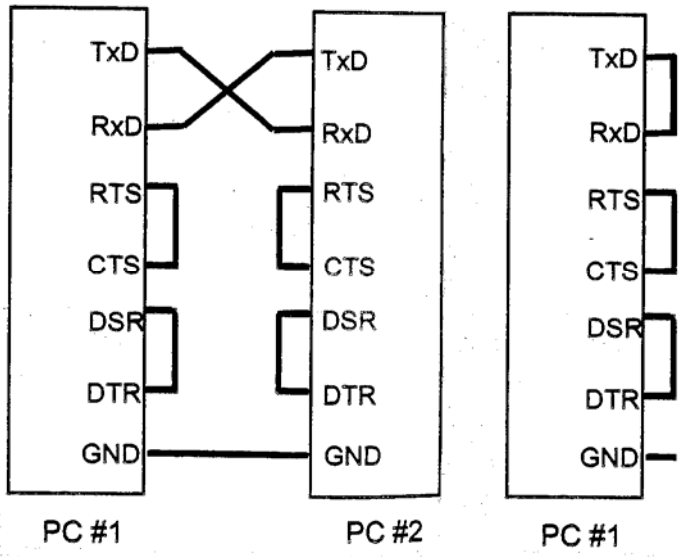
Due to this fact IC chips called line drivers and receivers must be used to perform this conversion. See diagram below.



The use of the PC's BIOS interrupts will make this experiment much easier. The first one is interrupt 14H, which will allow the user to access the COM ports. This interrupt can be used to initialize the COM ports, to read a value received from another device, and to write a value to be transmitted to another device through the COM ports. This interrupt is described at the end of this chapter. The second one is interrupt 16H, and this interrupt allows the user to directly access the keyboard. Function 0 checks the keyboard buffer for a character, if a character is available it returns its scan code in register AH and its ASCII code in register AL, and if a character is not available, it waits for a key press to happen then it returns the scan and ASCII codes. Function 1, which is similar to function 0, checks the keyboard buffer for a character, if a character is available it returns its scan code in register AH and its ASCII code in register AL, and then it sets the ZF = 0. If a character is not available, it sets ZF = 1, and it does not wait for a key press to happen. This interrupts are accessed by a software interrupt call, just like the interrupts used in some previous experiments. Several tables describing the interrupt's functions and PC's scan codes are given below.

Procedure

Write a program to send and receive information through the serial port of the PC. This program should display the transmitted information on rows 13 through 23 of the screen, with light blue characters on black background, and display the received data on rows 1 through 11 with yellow characters on black background. Use the diagram shown below to connect one PC to another, or to connect a PC in a loop-back mode to send and receive the information it just sent. The pin numbers for the DB9 and DB25 connectors are found in tables below.


TxD
RxD
RTS
CTS
DSR
DTR
GND
PC #1
TxD
RxD
RTS
CTS
DSR
DTR
GND
PC #2
TxD
RxD
RTS
CTS
DSR
DTR
GND
PC #1

DB25 pin numbers.

DB9 pin numbers.

AH **INT 14H Function**

00 **Initialize COM Port**

Additional Call Registers	Result Registers
AL = parameter (see below)	AH = port status (see below)
DX = port number (0 if COM1, 1 if COM2, etc.)	AL = modem status (see below)

Note 1: The parameter byte in AL is defined as follows

7 6 5 4 3 2 1 0	Indicates
x x x	Baud rate (000=110, 001=150, 010=300, 011=600, 100=1200, 101=2400, 110=4800, 111=9600)
x x	Parity (01=odd, 11=even, x0=none)
x	Stop bits (0 = 1, 1 = 2)
x x	Word length (10=7 bits, 11=8 bits)

Note 2: The port status returned in AH is defined as follows

7 6 5 4 3 2 1 0	Indicates
1	Timed-out
1	Transmit shift register empty
1	Transmit holding register empty
1	Break detected
1	Framing error detected
1	Parity error detected
1	Overrun error detected
1	Received data ready

Note 3: The modem status returned in AL is defined as follows

7 6 5 4 3 2 1 0	Indicates
1	Received line signal detect
1	Ring indicator
1	DSR (data set ready)
1	CTS (clear to send)
1	Change in receive line signal detect
1	Trailing edge ring indicator
1	Change in DSR status
1	Change in CTS status

AH **INT 14H Function**

01 **Write Character to COM Port**

Additional Call Registers	Result Registers
AL = character	AH bit 7 =0 if successful, 1 if not
DX = port number (0 if COM1, 1 if COM2, etc.)	AH bits 0 - 6 = status if successful
	AL = character

Note: The status byte in AH, bits 0 - 6, after the call is as follows:

6 5 4 3 2 1 0	Indicates
1	Transmit shift register empty
1	Transmit holding register empty
1	Break detected
1	Framing error detected
1	Parity error detected
1	Overrun error detected
1	Receive data ready

02 **Read Character from COM Port**

Additional Call Registers	Result Registers
DX = port number (0 if COM1, 1 if COM2, etc.)	AH bit 7 =0 if successful, 1 if not
	AH bits 0 - 6 = status if successful
	AL = character read

Note: The status byte in AH, bits 1 - 4, after the call is as follows:

4 3 2 1	Indicates
1	Break detected
1	Framing error detected
1	Parity error detected
1	Overrun error detected

03 **Read COM Port Status**

Additional Call Registers	Result Registers
DX = port number (0 if COM1, 1 if COM2, etc.)	AH =port status
	AL =modem status

Note: The port status and modem status returned in AH and AL are the same format as in INT 14H function 00H, described above.

SECTION E.6: INT 16H -- KEYBOARD

AH	Function

00H **Keyboard read**

Additional Call Registers	Result Registers
None	AH = key scan code AL = ASCII char

Note: Reads one character from the keyboard buffer and updates the head pointer.

01H **Get keyboard status**

Additional Call Registers	Result Registers
None	If no key waiting, ZF = 1. If key waiting, ZF = 0, AH = key scan code, AL = ASCII char.

Note: If a key is waiting, the scan code and character are returned in AH and AL, but the head pointer of the keyboard buffer is not updated.

Hex	Key	Hex	Key	Hex	Key	Hex	Key
01	Esc	15	Y and y	29	~ and ‘	3D	F3
02	! and 1	16	U and u	2A	LeftShift	3E	F4
03	@ and 2	17	I and i	2B	| and \	3F	F5
04	# and 3	18	O and o	2C	Z and z	40	F6
05	$ and 4	19	P and p	2D	X and x	41	F7
06	% and 5	1A	{ and [	2E	C and c	42	F8
07	^ and 6	1B	} and]	2F	V and v	43	F9
08	& and 7	1C	enter	30	B and b	44	F10
09	* and 8	1D	ctrl	31	N and n	45	NumLock
0A	(and 9	1E	A and a	32	M and m	46	ScrollLock
0B	) and 0	1F	S and s	33	< and ,	47	7 and Home
0C	_ and -	20	D and d	34	> and .	48	8 and UpArrow
0D	+ and =	21	F and f	35	? and /	49	9 and PgUp
0E	backspace	22	G and g	36	RightShift	4A	- (gray)
0F	tab	23	H and h	37	PrtSc and *	4B	4 and LeftArrow
10	Q and q	24	J and j	38	Alt	4C	5 (keypad)
11	W and w	25	K and k	39	Spacebar	4D	6 and RightArrow
12	E and e	26	L and l	3A	CapsLock	4E	+ (gray)
13	R and r	27	: and ;	3B	F1	4F	1 and End
14	T and t	28	" and ’	3C	F2	50	2 and DownArrow
						51	3 and PgDn
						52	0 and Ins
						53	. and Del

(Reprinted by permission from "IBM BIOS Technical Reference" c. 1987 by International Business Machines Corporation)

Experiment 10

Appendix A – Datasheets

1. Device datasheets are easily found by searching the internet with your favorite search engine. Search for the part number, i.e., 74LS08.

2. Request the TTL Logic book from the Monitor in room 128. You will need to present your student id to be able to use the book.

Appendix B – Parallel Port Information

Table 1

Data Register (Base Address)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	2	Data bit 0	No	Output
1	3	Data bit 1	No	Output
2	4	Data bit 2	No	Output
3	5	Data bit 3	No	Output
4	6	Data bit 4	No	Output
5	7	Data bit 5	No	Output
6	8	Data bit 6	No	Output
7	9	Data bit 7	No	Output
Status Register (Base Address + 1)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
3	15	nError	No	Input
4	13	Select	No	Input
5	12	PaperEnd	No	Input
6	10	nAck	No	Input
7	11	Busy	Yes	Input
Control Register (Base Address + 2)				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
0	1	NStrobe	Yes	Output
1	14	nAutoLF	Yes	Output
2	16	Ninit	No	Output
3	17	nSelectIn	Yes	Output
4		IRQ 1 = enabled		
5		Bidirectional 1 = input		
Ground Connections				
Bit	**Pin: DB-25**	**Signal Name**	**Inverted at connector?**	**I/O**
	18 – 25	Ground		

DB-25 and DB-9 connector pinout.

The "o" represent holes, the "." represent pins.

<table>
<tr><th colspan="2">DB-25 Connector</th></tr>
<tr><td>Connector 1 (Female)</td><td>Connector 2 (Male)</td></tr>
<tr><td><pre>
 13 <-------------------- 1

 ____
\ o o o o o o o o o o o o o /
 \ o o o o o o o o o o o o /
  ----
  25 <----------------- 14
</pre></td><td><pre>
  1 --------------------> 13

 ____
\ . . . . /
 \ . . . . /
  ----
  14 -----------------> 25
</pre></td></tr>
<tr><th colspan="2">DB-9 Connector</th></tr>
<tr><td>Connector 3 (Female)</td><td>Connector 4 (Male)</td></tr>
<tr><td><pre>
  5 4 3 2 1

 ____
\ o o o o o /
 \ o o o o /
  ----
   9 8 7 6
</pre></td><td><pre>
  1 2 3 4 5

 ____
\ . . . . . /
 \ . . . . /
  ----
   6 7 8 9
</pre></td></tr>
</table>

Each diagram shown above is the view you see when you look into the end of the cable.

Base addresses

- **278H**
- **378H**
- **3BCH**

Appendix C – Serial Port Information

DB9 pinout

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

DB25 pinout

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