UNIT 4 ASSEMBLY LANGUAGE PROGRAMMING (PART-II)

Structure **Page No.**

4.0	Introduction	77
4.1	Objectives	77
4.2	Use of Arrays in Assembly	77
4.3	Modular Programming	80
	4.3.1 The stack	
	4.3.2 FAR and NEAR Procedures	
	4.3.3 Parameter Passing in Procedures	
	4.3.4 External Procedures	
4.4	Interfacing Assembly Language Routines to High Level Language Programs	93
	4.4.1 Simple Interfacing	
	4.4.2 Interfacing Subroutines With Parameter Passing	
4.5	Interrupts	97
4.6	Device Drivers in Assembly	99
4.7	Summary	101
4.8	Solutions/ Answers	102

4.0 INTRODUCTION

In the previous units, we have discussed the instruction set, addressing modes, and other tools, which are needed to develop assembly language programs. We shall now use this knowledge in developing more advanced tools. We have divided this unit broadly into four sections. In the first section, we discuss the design of some simple data structures using the basic data types. Once the programs become lengthier, it is advisable to divide them into small modules, which can be easily written, tested and debugged. This leads to the concept of modular programming, and that is the topic of our second section in this unit. In the third section, we will discuss some techniques to interface assembly language programs to high level languages. We have explained the concepts using C and C ++ as they are two of the most popular high-level languages. In the fourth section we have designed some tools necessary for interfacing the microprocessor with external hardware modules.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- implement simple data structures in assembly language;
- write modular programs in assembly language;
- interface assembly program to high level language program; and
- analyse simple interrupt routines.

4.2 USE OF ARRAYS IN ASSEMBLY

An array is referencing using a base array value and an index. To facilitate addressing in arrays, 8086 has provided two index registers for mathematical computations, viz. BX and BP. In addition two index registers are also provided for string processing, viz. SI and DI. In addition to this you can use any general purpose register also for indexing.

An important application of array is the tables that are used to store related information. For example, the names of all the students in the class, their CGPA, the list of all the books in the library, or even the list of people residing in a particular area can be stored in different tables. An important application of tables would be character translation. It can be used for data encryption, or translation from one data type to another. A critical factor for such kind of applications is the speed, which just happens to be a strength of assembly language. The instruction that is used for such kind of applications is XLAT.

Let us explain this instruction with the help of an example:

Example 1:

Let us assume a table of hexadecimal characters representing all 16 hexadecimal digits in table:

```
HEXA    DB    '0123456789ABCDEF'
```

The table contains the ASCII code of each hexadecimal digit:

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Contents	30	31	32	33	34	35	36	37	38	39	41	42	43	44	45	46

(all value in hexadecimal)

If we place 0Ah in AL with the thought of converting it to ASCII, we need to set BX to the offset of HEXA, and invoke XLAT. You need not specify the table name with XLAT because it is implicitly passed by setting BX to the HEXA table offset. This instruction will do the following operations:

It will first add BX to AL, generating an effective address that points to the eleventh entry in the HEXA table.

The content of this entry is now moved to the AL register, that is, 41h is moved to AL.

In other words, XLAT sets AL to 41h because this value is located at HEXA table offset 0Ah. Please note that the 41h is the ASCII code for hex digit A. The following sequence of instructions would accomplished this:

```
MOV   AL,   0Ah                    ; index value
MOV   BX,   OFFSET HEXA            ; offset of the table HEXA
XLAT
```

The above tasks can be done without XLAT instruction but it will require a long series of instructions such as:

```
MOV      AL, 0Ah             ; index value
MOV      BX, OFFSET HEXA     ; offset of the table HEXA
PUSH     BX                  ; save the offset
ADD      BL, AL              ; add index value to table
                             ; HEXA offset
MOV      AL, [BX]            ; retrieve the entry
POP      BX                  ; restore BX
```

Let us use the instruction XLAT for data encoding. When you want to transfer a message through a telephone line, then such encoding may be a good way of preventing other users from reading it. Let us show a sample program for encoding.

PROGRAM 1:

; A program for encoding ASCII Alpha numerics.

```
; ALGORITHM:
        ; create the code table
        ; read an input string character by character
        ; translate it using code table
        ; output the strings

DATA    SEGMENT
        CODETABLE   DB 48 DUP (0)       ; no translation of first
                                        ; 48 ASCII
                    DB '4590821367'     ; ASCII codes 48 –
                                        ; 57    (30h – 39h)
                    DB 7 DUP (0)        ; no translation of
                                         these 7 characters
                    DB  'GVHZUSOBMIKPJCADLFTYEQNWXR'
                    DB 6 DUP (0)        ; no translation
                    DB  'gvhzusobmikpjcadlftyeqnwxr'
                    DB 133 DUP (0)      ; no translation of remaining
                                        ; character
DATA        ENDS

CODE    SEGMENT
        MOV     AX, DATA
        MOV     DS, AX              ; initialize DS
        MOV     BX, OFFSET CODETABLE        ; point to lookup table
GETCHAR:
        MOV     AH, 06              ; console input no wait
        MOV     DL, 0FFh            ; specify input request
        INT     21h                 ; call DOS
        JZ      QUIT                ; quit if no input is waiting
        MOV     DL, AL              ; save character in DL
        XLAT    CODETABLE           ; translate the character
        CMP     AL, 0               ; translatable?
        JE      PUTCHAR             ; no : write it as is.
        MOV     DL, AL              ; yes : move new character
                                    ; to DL
PUTCHAR:
        MOV     AH, 02              ; write DL to output
        INT     21h
        JMP     GETCHAR             ; get another character
QUIT:   MOV     AX, 4C00h
        INT     21h
CODE    ENDS
END
```

Discussion:

The program above will code the data. For example, a line from an input file will be encoded:

A SECRET Message	(Read from an input file)
G TUHFUY Juttgou	(Encoded output)

The program above can be run using the following command line. If the program file name is coding.asm

coding infile > outfile

The infile is the input data file, and outfile is the output data file.
You can write more such applications using 8086 assembly tables.

☞ Check Your Progress 1

1. Write a program to convert all upper case letters to lower case.

   ....

   ....

   ....

2. **State True or False** [T | F]

   a. Table handling cannot be done without using XLAT instruction. [ ]

   b. In XLAT instruction AX register contains the address of the first entry of the table. [ ]

   c. In XLAT instruction the desired element value is returned in AL register. [ ]

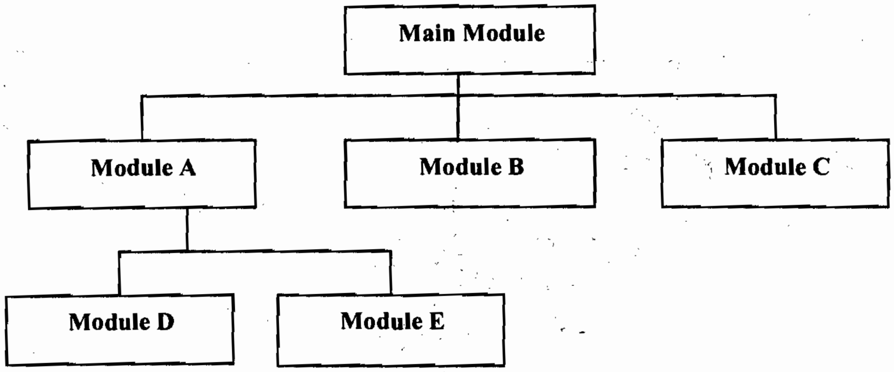
4.3 MODULAR PROGRAMMING

Modular programming refers to the practice of writing a program as a series of independently assembled source files. Each source file is a modular program designed to be assembled into a separate object file. Each object file constitutes a module. The linker collects the modules of a program into a coherent whole.

There are several reasons a programmer might choose to modularise a program.

1. Modular programming permits breaking a large program into a number of smaller modules each of more manageable size.
2. Modular programming makes it possible to link source code written in two separate languages. A hybrid program written partly in assembly language and partly in higher level language necessarily involves at least one module for each language involved.
3. Modular programming allows for the creation, maintenance and reuse of a library of commonly used modules.
4. Modules are easy to comprehend.
5. Different modules can be assigned to different programs.
6. Debugging and testing can be done in a more orderly fashion.
7. Document action can be easily understood.
8. Modifications may be localised to a module.

A modular program can be represented using hierarchical diagram:



The advantages of modular programming are:

1. Smaller, easier modules to manage
2. Code repetition may be avoided by reusing modules.

You can divide a program into subroutines or procedures. You need to CALL the procedure whenever needed. A subroutine call transfers the control to subroutine instructions and brings the control back to calling program.

4.3.1 The Stack

A procedure call is supported by a stack. So let us discuss stack in assembly. Stacks are Last In First Out data structures, and are used for storing the return addresses of the procedures and for parameter passing and saving the return value.

In 8086 microprocessor a stack is created in the stack segment. The SS register stores the offset of stack segment and SP register stores the top of the stack. A value is pushed in to top of the stack or taken out (poped) from the top of the stack. The stack segment can be initialized as follows:

```
STACK_ SEG SEGMENT STACK
      DW 100          DUP (0)
      TOS LABEL       WORD
STACK_SEG ENDS

CODE SEGMENT
        ASSUME CS:CODE, SS:STACK_SEG
        MOV   AX, STACK_SEG
        MOV   SS,AX               ; initialise stack segment
        LEA   SP,TOP              ; initialise stack pointer
CODE  ENDS
        END
```

The directive STACK_SEG SEGMENT STACK declares the logical segment for the stack segment. DW 100 DUP(0) assigns actual size of the stack to 100 words. All locations of this stack are initialized to zero. The stacks are identified by the stack top and that is why the Label Top of Stack (TOS) has been selected. Please note that the stack in 8086 is a WORD stack. Stack facilities involve the use of indirect addressing through a special register, the stack pointer (SP). SP is automatically decremented as items are put on the stack and incremented as they are retrieved. Putting something on to stack is called a PUSH and taking it off is called a POP. The address of the last element pushed on to the stack is known as the top of the stack (TOS).

Name	Mnemonics	Description
Push onto the stack	PUSH SRC	SP← SP – 2 SP+1 and SP location are assign the SRC
Pop from the stack	POP DST	DST is a assigned values stored at stack top SP← SP + 2

4.3.2 Far and Near Procedures

Procedure provides the primary means of breaking the code in a program into modules. Procedures have one major disadvantage, that is, they require extra code to join them together in such a way that they can communicate with each other. This extra code is sometimes referred to as linkage overhead.

A procedure call involves:

1. Unlike other branch instructions, a procedure call must save the address of the next instruction so that the return will be able to branch back to the proper place in the calling program.

2. The registers used by the procedures need to be stored before their contents are changed and then restored just before the procedure is finished.

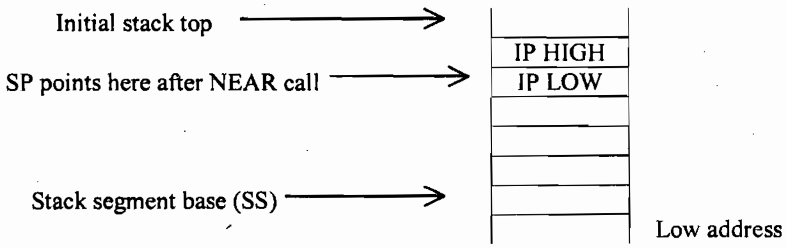
3. A procedure must have a means of communicating or sharing data with the procedures that call it, that is parameter passing.

Calls, Returns, and Procedures definitions in 8086

The 8086 microprocessor supports CALL and RET instructions for procedure call.

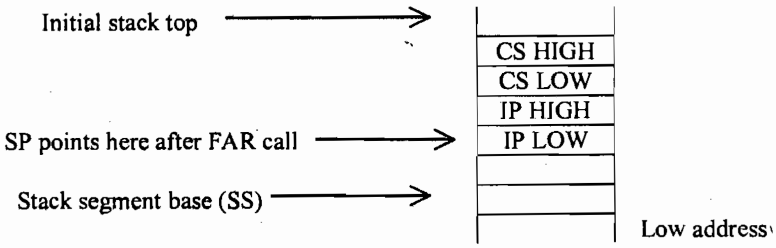
The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. In addition, it also initialized IP with the address of the procedure. The RET instructions simply pops the return address from the stack. 8086 supports two kinds of procedure call. These are FAR and NEAR calls.

The NEAR procedure call is also known as Intrasegment call as the called procedure is in the same segment from which call has been made. Thus, only IP is stored as the return address. The IP can be stored on the stack as:



Please note the growth of stack is towards stack segment base. So stack becomes full on an offset 0000h. Also for push operation we decrement SP by 2 as stack is a word stack (word size in 8086 = 16 bits) while memory is byte organised memory.

FAR procedure call, also known as intersegment call, is a call made to separate code segment. Thus, the control will be transferred outside the current segment. Therefore, both CS and IP need to be stored as the return address. These values on the stack after the calls look like:



When the 8086 executes the FAR call, it first stores the contents of the code segment register followed by the contents of IP on to the stack. A RET from the NEAR procedure. Pops the two bytes into IP. The RET from the FAR procedure pops four bytes from the stack.

Procedure is defined within the source code by placing a directive of the form:

<Procedure name> PROC <Attribute>

A procedure is terminated using:

<Procedure name> ENDP

The <procedure name> is the identifier used for calling the procedure and the <attribute> is either NEAR or FAR. A procedure can be defined in:

1. The same code segment as the statement that calls it.
2. A code segment that is different from the one containing the statement that calls it, but in the same source module as the calling statement.
3. A different source module and segment from the calling statement.

In the first case the <attribute> code NEAR should be used as the procedure and code are in the same segment. For the latter two cases the <attribute> must be FAR.

Let us describe an example of procedure call using NEAR procedure, which contains a call to a procedure in the same segment.

PROGRAM 2:

Write a program that collects in data samples from a port at 1 ms interval. The upper 4 bits collected data same as mastered and stored in an array in successive locations.

```
; REGISTERS       :Uses CS, SS, DS, AX, BX, CX, DX, SI, SP
; PROCEDURES      : Uses WAIT

DATA_SEG     SEGMENT
       PRESSURE         DW       100      DUP(0)      ; Set up array of 100 words
       NBR_OF_SAMPLES            EQU      100
       PRESSURE_PORT EQU         0FFF8h               ; hypothetical input port
DATA_SEG     ENDS

STACK_SEG SEGMENT STACK
       DW         40             DUP(0)               ; set stack of 40 words
       STACK_TOP   LABEL WORD
STACK_SEG   ENDS

CODE_SEG    SEGMENT
      ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:      MOV     AX, DATA_SEG            ; Initialise data segment register
            MOV     DS, AX
            MOV     AX, STACK_SEG           ; Initialise stack segment register
            MOV     SS, AX
            MOV     SP, OFFSET STACK – TOP  ; initialise stack pointer  top of
                                            ; stack
            LEA     SI, PRESSURE            ; SI points to start of array
                                            ; PRESSURE
            MOV     BX, NBR_OF_SAMPLES      ; Load BX with number
                                            ; of samples
            MOV     DX, PRESSURE_PORT       ; Point DX at input port
                                            ; it can be any A/D converter or
                                            ; data port.

READ_NEXT: IN          AX, DX           ; Read data from port
                                        ; please note use of IN instruction
            AND        AX, 0FFFH        ; Mask upper 4 bits of AX
            MOV        [SI], AX         ; Store data word in array
            CALL       WAIT             ; call procedures wait for delay
```

```
            INC       SI             ; Increment SI by two as dealing with
            INC       SI             ; 16 bit words and not bytes
            DEC       BX             ; Decrement sample counter
            JNZ       READ_NEXT      ; Repeat till 100
                                     ; samples are collected
STOP:       NOP
WAIT        PROC      NEAR
            MOV       CX, 2000H      ; Load delay value
                                     ; into CX
HERE:       LOOP      HERE           ; Loop until CX = 0
            RET
WAIT        ENDP
CODE_SEG    ENDS
            END
```

Discussion:

Please note that the CALL to the procedure as above does not indicate whether the call is to a NEAR procedure or a FAR procedure. This distinction is made at the time of defining the procedure.

The procedure above can also be made a FAR procedure by changing the definition of the procedure as:

```
WAIT        PROC FAR
            .
            .
WAIT        ENDS
```

The procedure can now be defined in another segment if the need so be, in the same assembly language file.

4.3.3 Parameter Passing in Procedures

Parameter passing is a very important concept in assembly language. It makes the assembly procedures more general. Parameter can be passed to and from to the main procedures. The parameters can be passed in the following ways to a procedure:

1. Parameters passing through registers
2. Parameters passing through dedicated memory location accessed by name
3. Parameters passing through pointers passed in registers
4. Parameters passing using stack.

Let us discuss a program that uses a procedure for converting a BCD number to binary number.

PROGRAM 3:

Conversion of BCD number to binary using a procedure.

Algorithm for conversion procedure:

Take a packed BCD digit and separate the two digits of BCD.
Multiply the upper digit by 10 (0Ah)
Add the lower digit to the result of multiplication

The implementation of the procedure will be dependent on the parameter-passing scheme. Let us demonstrate this with the help of three programs.

Program 3 (a): Use of registers for parameter passing: This program uses AH register for passing the parameter.

We are assuming that data is available in memory location. BCD and the result is stored in BIN

```
;REGISTERS           : Uses CS, DS, SS, SP, AX
;PROCEDURES          : BCD-BINARY

DATA_SEG             SEGMENT
         BCD         DB 25h                    ; storage for BCD value
         BIN         DB ?                      ; storage for binary value
DATA_SEG             ENDS
STACK_SEG            SEGMENT     STACK
                     DW          200 DUP(0)         ; stack of 200 words
         TOP_STACK   LABEL       WORD
STACK_SEG            ENDS

CODE_SEG             SEGMENT
         ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:   MOV         AX, DATA_SEG              ; Initialise data segment
         MOV         DS, AX                    ; Using AX register
         MOV         AX, STACK_SEG             ; Initialise stack
         MOV         SS, AX                    ; Segment register. Why
                                               ; stack?
         MOV         SP, OFFSET TOP_STACK      ; Initialise stack pointer
         MOV         AH, BCD
         CALL        BCD_BINARY                ; Do the conversion
         MOV         BIN, AH                   ; Store the result in the
                                               ; memory
                     :
                     :
; Remaining program can be put here
;PROCEDURE          : BCD_BINARY - Converts BCD numbers to binary.
;INPUT              : AH with BCD value
;OUTPUT             : AH with binary value
;DESTROYS           : AX

BCD_BINARY   PROC NEAR

       PUSHF                        ; Save flags
       PUSH     BX                  ; and registers used in procedure
       PUSH     CX                  ; before starting the conversion
                                    ; Do the conversion
       MOV      BH, AH              ; Save copy of BCD in BH
       AND      BH, 0Fh             ; and mask the higher bits. The lower digit
                                    ; is in BH
       AND      AH, 0F0h            ; mask the lower bits. The higher digit is in AH
                                    ; but in upper 4 bits.
       MOV      CH, 04              ; so move upper BCD digit to lower
       ROR      AH, CH              ; four bits in AH
       MOV      AL, AH              ; move the digit in AL for multiplication
       MOV      BH, 0Ah             ; put 10 in BH
       MUL      BH                  ; Multiply upper BCD digit in AL
                                    ; by 0Ah in BH, the result is in AL
       MOV      AH, AL              ; the maximum/ minimum number would not
                                    ; exceed 8 bits so move AL to AH
       ADD      AH, BH              ; Add lower BCD digit to MUL result
; End of conversion, binary result in AH
              POP       CX              ; Restore registers
              POP       BX
              POPF
```

```
                RET                         ; and return to calling program
BCD_BINARY      ENDP
CODE_SEG        ENDS
                END         START
```

Discussion:

The above program is not an optimum program, as it does not use registers minimally. By now you should be able to understand this module. The program copies the BCD number from the memory to the AH register. The AH register is used as it is in the procedure. Thus, the contents of AH register are used in calling program as well as procedure; or in other words have been passed from main to procedure. The result of the subroutine is also passed back to AH register as returned value. Thus, the calling program can find the result in AH register.

The advantage of using the registers for passing the parameters is the ease with which they can be handled. The disadvantage, however, is the limit of parameters that can be passed. For example, one cannot pass an array of 100 elements to a procedure using registers.

Passing Parameters in General Memory

The parameters can also be passed in the memory. In such a scheme, the name of the memory location is used as a parameter. The results can also be returned in the same variables. This approach has a severe limitation. It is that you will be forced to use the same memory variable with that procedure. What are the implications of this bound? Well in the example above we will be bound that variable BCD must contain the input. This procedure cannot be used for a value stored in any other location. Thus, it is a very restrictive method of procedural call.

Passing Parameters Using Pointers

This method overcomes the disadvantage of using variable names directly in the procedure. It uses registers to pass the procedure pointers to the desired data. Let us explain it further with the help of a newer version of the last program.

Program 3 (c) version 2:

```
DATA_SEG        SEGMENT
                BCD         DB      25h     ; Storage for BCD test value
                BIN         DB      ?       ; Storage for binary value
DATA_SEG        ENDS

STACK_SEG SEGMENT    STACK
                     DW      100 DUP(0)     ; Stack of 100 words
      TOP_STACK LABEL    WORD
STACK_SEG  ENDS

CODE_SEG     SEGMENT
             ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:       MOV     AX, DATA_SEG           ; Initialize data
             MOV     DS, AX                 ; segment using AX register
             MOV     AX, STACK_SEG          ; initialize stack
             MOV     SS, AX                 ; segment. Why stack?
             MOV     SP, OFFSET TOP_STACK   ; initialize stack pointer
; Put pointer to BCD storage in SI and DI prior to procedure call.
             MOV     SI, OFFSET BCD         ; SI now points to BCD_IN
             MOV     DI, OFFSET BIN         ; DI points BIN_VAL
                                            ; (returned value)
             CALL    BCD_BINARY             ; Call the conversion
```

```
                                        ; procedure
        NOP                             ; Continue with program
                                        ; here

; PROCEDURE        : BCD_BINARY Converts BCD numbers to binary.
; INPUT            : SI points to location in memory of data
; OUTPUT           : DI points to location in memory for result
; DESTROYS         : Nothing

BCD_BINARY   PROC      NEAR
        PUSHF                       ; Save flag register
        PUSH      AX                ; and AX registers
        PUSH      BX                ; BX
        PUSH      CX                ; and CX
        MOV       AL, [SI]          ; Get BCD value from memory
                                    ; for conversion
        MOV       BL, AL            ; copy it in BL also
        AND       BL, 0Fh           ; and mask to get lower 4 digits
        AND       AL, 0F0h          ; Separate upper 4 bits in AL
        MOV       CL, 04            ; initialize counter CL so that upper digit
        ROR       AL, CL            ; in AL can be brought to lower 4 bit
                                    ; positions in AL
        MOV       BH, 0Ah           ; Load 10 in BH
        MUL       BH                ; Multiply upper digit in AL by 10
                                    ; The result is stored in AL
        ADD       AL, BL            ; Add lower BCD digit in BL to result of
                                    ; multiplication
; End of conversion, now restore the original values prior to call. All calls will be in
; reverse order to save above. The result is in AL register.
        MOV       [DI], AL          ; Store binary value to memory
        POP       CX                ; Restore flags and
        POP       BX                ; registers
        POP       AX
        POPF
        RET
BCD_BINARY        ENDP
CODE_SEG          ENDS
        END       START
```

Discussion:

In the program above, SI points to the BCD and the DI points to the BIN. The instruction MOV AL,[SI] copies the byte pointed by SI to the AL register. Likewise, MOV [DI], AL transfers the result back to memory location pointed by DI.

This scheme allows you to pass the procedure pointers to data anywhere in memory. You can pass pointer to individual data element or a group of data elements like arrays and strings. This approach is used for parameters passing to BIOS procedures.

Passing Parameters Through Stack

The best technique for parameter passing is through stack. It is also a standard technique for passing parameters when the assembly language is interfaced with any high level language. Parameters are pushed on the stack and are referenced using BP register in the called procedure. One important issue for parameter passing through stack is to keep track of the stack overflow and underflow to keep a check on errors. Let us revisit the same example, but using stack for parameter passing.

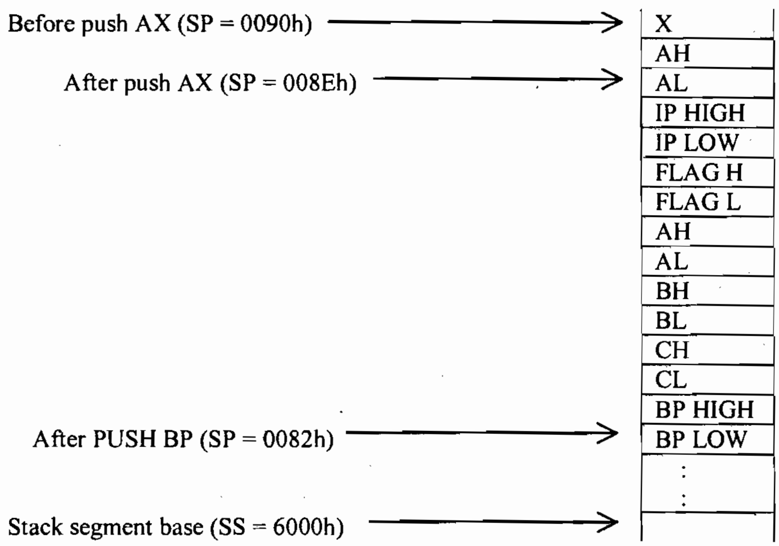
PROGRAM 3: Version 3

```
DATA_SEG     SEGMENT
             BCD          DB      25h       ; Storage for BCD test value
             BIN          DB      ?         ; Storage for binary value
DATA_SEG     ENDS

STACK_SEG    SEGMENT      STACK
             DW           100 DUP(0)        ; Stack of 100 words
             TOP_STACK  LABEL   WORD
STACK_SEG    ENDS

CODE_SEG     SEGMENT
        ASSUME CS:CODE_SEG, DS:DATA_SEG, SS:STACK_SEG
START:       MOV      AX, DATA                 ; Initialise data segment
             MOV      DS, AX                   ; using AX register
             MOV      AX, STACK-SEG .          ; initialise stack segment
             MOV      SS, AX                   ; using AX register
             MOV      SP, OFFSET TOP_STACK     ; initialise stack pointer
             MOV      AL, BCD                  ; Move BCD value into AL
             PUSH     AX                       ; and push it onto word stack
             CALL     BCD_BINARY               ; Do the conversion
             POP      AX                       ; Get the binary value
             MOV      BIN, AL                  ; and save it
             NOP                               ; Continue with program
; PROCEDURE           : BCD_BINARY Converts BCD numbers to binary.
; INPUT               : None - BCD value assumed to be on stack before call
; OUTPUT              : None - Binary value on top of stack after return
; DESTROYS            : Nothing

BCD_BINARY   PROC     NEAR
             PUSHF                             ; Save flags
             PUSH     AX                       ; and registers : AX
             PUSH     BX                       ; BX
             PUSH     CX                       ; CX
             PUSH     BP                       ; BP. Why BP?
             MOV      BP, SP                   ; Make a copy of the
                                               ; stack pointer in BP
             MOV      AX, [BP+ 12]             ; Get BCD number from
                                               ; stack. But why it is on
; BP+12 location? Please note 5 PUSH statements + 1 call which is intra-segment (so
; just IP is stored) so total 6 words are pushed after AX has been pushed and since it is
; a word stack so the BCD value is stored on 6 × 2 = 12 locations under stack. Hence
; [BP + 12] (refer to the figure given on next page).
             MOV      BL, AL          ; Save copy of BCD in BL
             AND      BL, 0Fh         ; mask lower 4 bits
             AND      AL, F0H         ; Separate upper 4 bits
             MOV      CL, 04          ; Move upper BCD digit to low
             ROR      AL, CL          ; position BCD digit for multiply location
             MOV      BH, 0Ah           ; Load 10 in BH
             MUL      BH                ; Multiply upper BCD digit in AL by 10
                                        ; the result is in AL
             ADD      AL, BL            ; Add lower BCD digit to result.
             MOV      [BP + 12], AX     ; Put binary result on stack
             ; Restore flags and registers
             POP      BP
             POP      CX
             POP      BX
             POP      AX
```

```
                POPF
                RET
BCD_BINARY              ENDP
CODE_SEG                ENDS
END         START
```

Discussion:

The parameter is pushed on the stack before the procedure call. The procedure call causes the current instruction pointer to be pushed on to the stack. In the procedure flags, AX, BX, CX and BP registers are also pushed in that order. Thus, the stack looks to be:



The instruction MOV BP, SP transfers the contents of the SP to the BP register. Now BP is used to access any location in the stack, by adding appropriate offset to it. For example, MOV AX, [BP + 12] instruction transfers the word beginning at the 12th byte from the top of the stack to AX register. It does not change the contents of the BP register or the top of the stack. It copies the pushed value of AH and AL at offset 008Eh into the AX register. This instruction is not equivalent to POP instruction.

Stacks are useful for writing procedures for multi-user system programs or recurvise procedures. It is a good practice to make a stack diagram as above while using procedure call through stacks. This helps in reducing errors in programming.

4.3.4 External Procedures

These procedures are written and assembled in separate assembly modules, and later are linked together with the main program to form a bigger module. Since the addresses of the variables are defined in another module, we need segment combination and global identifier directives. Let us discuss them briefly.

Segment Combinations

In 8086 assembler provides a means for combining the segments declared in different modules. Some typical combine types are:

1. PUBLIC: This combine directive combines all the segments having the same names and class (in different modules) as a single combined segment.
2. COMMON: If the segments in different object modules have the same name and the COMMON combine type then they have the same beginning address. During execution these segments overlay each other.

3. STACK: If the segments in different object modules have the same name and the combine type is STACK, then they become one segment, with the length the sum of the lengths of individual segments.

These details will be more clear after you go through program 4 and further readings.

Identifiers

a) **Access to External Identifiers:** An external identifier is one that is referred in one module but defined in another. You can declare an identifier to be external by including it on as EXTRN in the modules in which it is to be referred. This tells the assembler to leave the address of the variable unresolved. The linker looks for the address of this variable in the module where it is defined to be PUBLIC.
b) **Public Identifiers:** A public identifier is one that is defined within one module of a program but potentially accessible by all of the other modules in that program. You can declare an identifier to be public by including it on a PUBLIC directive in the module in which it is defined.

Let us explain all the above with the help of the following example:

PROGRAM 4:

Write a procedure that divides a 32-bit number by a 16-bit number. The procedure should be general, that is, it is defined in one module, and can be called from another assembly module.

```
; REGISTERS            :Uses CS, DS, SS, AX, SP, BX, CX
; PROCEDURES           : Far Procedure SMART_DIV
DATA_SEG    SEGMENT     WORD    PUBLIC
            DIVIDEND    DW      2345h, 89AB     ; Dividend =
                                                ; 89AB2345H
            DIVISOR     DW      5678H           ; 16-bit divisor
            MESSAGE     DB      'INVALID DIVIDE', '$'
DATA_SEG    ENDS

MORE_DATA  SEGMENT      WORD
      QUOTIENT          DW      2  DUP(0)
      REMAINDER         DW      0
MORE_DATA ENDS

STACK_SEG     SEGMENT   STACK
      DW                100 DUP(0)              ; Stack of 100 words
      TOP – STACK       LABEL     WORD          ; top of stack pointer
STACK_SEG     ENDS

PUBLIC        DIVISOR

PROCEDURES  SEGMENT   PUBLIC          ; SMART_DIVis declared as an
      EXTRN SMART_DIV: FAR            ; external label in procedure
                                      ; segment of  type FAR
PROCEDURES  ENDS
; declare the code segment as PUBLIC so that it can be merged with other PUBLIC
; segments
CODE_SEG    SEGMENT     WORD     PUBLIC
      ASSUME CS:CODE, DS:DATA_SEG, SS:STACK SEG
START:   MOV    AX, DATA_SEG               ; Initialize data segment
         MOV    DS, AX                     ; using AX register
         MOV    AX, STACK_SEG              ; Initialize stack segment
```

```
          MOV   SS, AX                     ; using AX register
          MOV   SP, OFFSET TOP_STACK       ; Initialize stack pointer
          MOV   AX, DIVIDEND               ; Load low word of
                                           ; dividend
          MOV   DX DIVIDEND + 2            ; Load high word of
                                           ; dividend
          MOV   CX, DIVISOR                ; Load divisor
          CALL SMART_DIV
; This procedure returns Quotient in the DX:AX pair and Remainder in CX register.
; Carry bit is set if result is invalid.
          JNC   SAVE_ALL                   ; IF carry = 0, result valid
          JMP   STOP                       ; ELSE carry set, don't
                                           ; save result
          ASSUME DS:MORE_DATA              ; Change data segment
SAVE_ALL:       PUSH DS                    ; Save old DS
          MOV   BX, MORE_DATA              ; Load new data segment
          MOV   DS, BX                     ; register
          MOV   QUOTIENT, AX               ; Store low word of
                                           ; quotient
          MOV   QUOTIENT + 2, DX           ; Store high word of
                                           ; quotient
          MOV   REMAINDER, CX              ; Store remainder
          ASSUME DS:DATA_SEG
          POP   DS                         ; Restore initial DS
          JMP   ENDING
STOP:     MOV   DL, OFFSET MESSAGE
          MOV   AX, AH 09H
          INT   21H
ENDING:   NOP
CODE_SEG        ENDS
          END   START
```

Discussion:

The linker appends all the segments having the same name and PUBLIC directive with segment name into one segment. Their contents are pulled together into consecutive memory locations.

The next statement to be noted is PUBLIC DIVISOR. It tells the assembler and the linker that this variable can be legally accessed by other assembly modules. On the other hand EXTRN SMART_DIV:FAR tells the assembler that this module will access a label or a procedure of type FAR in some assembly module. Please also note that the EXTRN definition is enclosed within the PROCEDURES SEGMENT PUBLIC and PROCEDURES ENDS, to tell the assembler and linker that the procedure SMART_DIV is located within the segment PROCEDURES and all such PROCEDURES segments need to be combined in one. Let us now define the PROCEDURE module:

```
; PROGRAM   MODULE      PROCEDURES

; INPUT       : Dividend - low word in AX, high word in DX, Divisor in CX
; OUTPUT      : Quotient  - low word in AX, high word in DX. Remainder in CX
              ; Carry     - carry flag is set if try to divide by zero
; DESTROYS    : AX, BX, CX, DX, BP, FLAGS
DATA_SEG    SEGMENT     PUBLIC    ; This block tells the assembler that
EXTRN DIVISOR:WORD                ; the divisor is a word variable and is
DATA_SEG     ENDS                 ; external to this procedure. It would be
                                  ; found in segment named DATA_SEG
PUBLIC        SMART_DIV           ; SMART_DIV is available to
                                  ; other modules. It is now being defined
```

```
                                          ; in PROCEDURES SEGMENT.
PROCEDURES    SEGMENT   PUBLIC
SMART_DIV     PROC      FAR
      ASSUME CS:PROCEDURES, DS:DATA_SEG
      CMP     DIVISOR, 0                  ; This is just to demonstrate the use of
                                          ; external variable, otherwise we can
                                          ; check it through CX register which
                                          ; contains the divisor.
      JE      ERROR_EXIT                  ; IF divisor = 0, exit procedure
      MOV     BX, AX                      ; Save low order of dividend
      MOV     AX, DX                      ; Position high word for lst divide
      MOV     DX, 0000h                   ;·Zero DX
      DIV     CX                          ; DX:AX/CX, quotient in AX,
                                          ; remainder in DX
      MOV     BP, AX                      ; transfer high order of final result to BP
      MOV     AX, BX                      ; Get back low order of dividend. Note
                                          ; DX contains remainder so DX : AX is
                                          ; the actual number
      DIV     CX                          ; DX:AX/CX, quotient in AX,
                                          ; 2nd remainder that is final remainder
                                          ; in DX
      MOV     CX, DX                      ; Pass final remainder in CX
      MOV     DX, BP                      ; Pass high order of quotient in DX
                                          ; AX contains lower word of quotient
      CLC                                 ; Clear carry to indicate valid result
      JMP     EXIT                        ; Finished
ERROR_EXIT: STC                           ; Set carry to indicate divide by zero
EXIT:   RET
SMART_DIV     ENDP
PROCEDURES    ENDS
            END
```

Discussion:

The procedure accesses the data item named DIVISOR, which is defined in the main, therefore the statement EXTRN DIVISOR:WORD is necessary for informing assembler that this data name is found in some other segment. The data type is defined to be of word type. Please not that the DIVISOR is enclosed in the same segment name as that of main that is DATA_SEG and the procedure is in a PUBLIC segment.

☞ Check Your Progress 2

T	F

1. State True or False

(a) A NEAR procedure can be called only in the segment it is defined. ☐

(b) A FAR call uses one word in the stack for storing the return address. ☐

(c) While making a call to a procedure, the nature of procedure that is NEAR or FAR must be specified. ☐

(d) Parameter passing through register is not suitable when large numbers of parameters are to be passed. ☐

(e) Parameter passing in general memory is a flexible way of passing parameters. ☐

(f) Parameter passing through pointers can be used to pass a group of data elements. ☐

(f) Parameter passing through stack is used whenever assembly language programs are interfaced with any high level language programs. ☐

(h) In multiuser systems parameters should be passed using pointers. ☐

(i) A variable say USAGE is declared in a PROCEDURE segment, however it is used in a separate module. In such a case the declaration of USAGE should contain EXTRN verb. ☐

(i) A segment if declared PUBLIC informs the linker to append all the segments with same name into one. ☐

2. Show the stack if the following statements are encountered in sequence.

a) Call to a NEAR procedure FIRST at 20A2h:0050h
b) Call to a FAR procedure SECOND at location 3000h:5055h
c) RETURN from Procedure FIRST.

4.4 INTERFACING ASSEMBLY LANGUAGE ROUTINES TO HIGH LEVEL LANGUAGE PROGRAMS

By now you can write procedures, both external and internal, and pass parameters, especially through stack, let us use these concepts, to see how assembly language can be interfaced to some high level language programs. It is very important to learn this concept, because then you can combine the advantages of both the types of languages, that is, the ease of programming of high level languages and the speed and the scope of assembly language. Assembly language can be interfaced with most of the high level languages like C, C + + and database management systems.

What are the main considerations for interfacing assembly to HLL? To answer that we need to answer the following questions:

- How is the subroutine invoked?
- How are parameters passed?
- How are the values returned?
- How do you declare various segments so that they are consistent across calling program?

The answer to the above questions are dependent on the high level language (HLL). Let us take C Language as the language for interfacing. The C Language is very useful for writing user interface programs, but the code produced by a C compiler does not execute fast enough for telecommunications or graphics applications. Therefore, system programs are often written with a combination of C and assembly language functions. The main user interface may be written in C and specialized high speed functions written in assembly language.

The guidelines for calling assembly subroutines from C are:

(i) Memory model: The calling program and called assembly programs must be defined with the same memory model. One of the most common convention that makes NEAR calls is .MODEL SMALL, C
(ii) The naming convention normally involve an underscore (_) character preceding the segment or function name. Please note, however, this underscore is not used while making a call from C function. Please be careful about case sensitivity.

You must give a specific segment name to the code segment of your assembly language subroutine. The name varies from compiler to compiler. Microsoft C, and Turbo C require the code segment name to be_TEXT or a segment name with suffix_TEXT. Also, it requires the segment name _DATA for the data segment.

(iii) The arguments from C to the assembly language are passed through the stack. For example, a function call in C:

function_name (arg1, arg2, ..., argn) ;

would push the value of each argument on the stack in the reverse order. That is, the argument *argn* is pushed first and *arg1* is pushed last on the stack. A value or a pointer to a variable can also be passed on the stack. Since the stack in 8086 is a word stack, therefore, values and pointers are stored as words on stack or multiples of the word size in case the value exceeds 16 bits.

(iv) You should remember to save any special purpose registers (such as CS, DS, SS, ES, BP, SI or DI) that may be modified by the assembly language routine. If you fail to save them, then you may have undesirable/ unexplainable consequences, when control is returned to the C program. However, there is no need to save AX, BX, CX or DX registers as they are considered volatile.

(v) Please note the compatibility of data types:

char	Byte (DB)
int	Word (DW)
long	Double Word (DD)

(vi) Returned value: The called assembly routine uses the followed registers for returned values:

char	AL
Near/ int	AX
Far/ long	DX : AX

Let us now look into some of the examples for interfacing.

4.4.1 Simple Interfacing

The following is a sample of the coding, used for procedure interfacing:

```
PUBLIC CUROFF
        _TEXT SEGMENT WORD PUBLIC 'CODE'
                ASSUME CS:_TEXT
                _CUROFF PROC NEAR     ; for small model
        :
        :
```

The same thing can be written using the newer simplified directives in the following manner:

```
        PUBLIC CUROFF
        .MODEL small,C
        .CODE
        CUROFF PROC
                :
                :
```

This second source code is much cleaner and easier to read. The directives .MODEL and .CODE instruct the assembler to make the necessary assumptions and adjustments so that the routine will work with a small model of C program. (Please refer to Assembler manuals on details on models of C program. The models primarily differ in number of segments).

PROGRAM 5:

Write an assembly function that hides the cursor. Call it from a C program.

```
                . PUBLIC CUROFF
                . MODEL small,C
                . CODE
CUROFF  PROC
        MOV     AH,3            ; get the current cursor position
        XOR     BX,BX           ; empty BX register
        INT     10h             ; use int 10hto do above
        OR      CH,20h          ; force to OFF condition
        MOV     AH,01           ; set the new cursor values
        INT     10h
        RET
CUROFF ENDP
                END
```

For details on various interrupt functions used in this program refer to further readings.

The C program to test this routine is as follows:

```
          # include < stdio.h
          void curoff(void);
          void main()
{
      printf("%s\n, "The cursor is now turning off);
      curoff();
}
```

You can write another procedure in assembly language program to put the cursor on. This can be done by replacing OR CH,20h instruction by AND CH,1Fh. You can call this new function from C program to put the cursor on after the curoff.

4.4.2 Interfacing Subroutines With Parameter Passing

Let us now write a C program that calls the assembly program for parameter passing. Let us extend the previous two programs such that if on function call 0 is passed then cursor is turned off and if 1 is passed then cursor is turned on. The calling C program for such may look like:

```
# include < stdio.h
void cursw(int);
void main()
{
        printf("%s\n", "the cursor is now turning off");
        cursw(0); /* call to assembly routine with 0 as parameter
        getchar();
        printf("%s\n","the cursor is now turning on");
        cursw(l); /* call to assembly routine with parameter as1.
}
```

The variables in C or C + + are passed on the stack.

Let us now write the assembly routine:

PROGRAM 6:

Write a subroutine in C for toggling the cursor using old directives.

```
;
; use small memory model for C – near code segment

_DATA       SEGMENT WORD    'DATA'
   CURVAL   EQU             [BP+4]          ; parameters
_DATA       ENDS

_TEXT       SEGMENT         BYTE  PUBLIC         'CODE'
DGROUP      GROUP           _DATA
            ASSUME          CS:_TEXT,       DS:DGROUP,  SS:DGROUP
            PUBLIC          _CURSW
_CURSW      PROC            NEAR
            PUSH            BP              ; BP register of caller is saved
            MOV             BP, SP          ; BP is pointing to stack now
            MOV             AX, CURVAL
            CMP             AX, 0H
            JZ              CUROFF          ; Execute code for cursor off
            CMP             AX, 01H
            JZ              CURON           ; Execute code for cursor on
            JMP             OVER            ; Error in parameter, do nothing
CUROFF:                                     ; write code for curoff
      :
      :
            JMP             OVER
CURON:                                      ; write code for curon
      :
      :
OVER:       POP     BP
            RET
_CURSW      ENDP
_TEXT       ENDS
            END
```

Why the parameter is found in [BP+4]? Please look into the following stack for the answer.

Parameter (0 or 1)	BP + 4
Return Address	BP + 2
Old value	BP + 0

PROGRAM 7:

Write a subroutine in C that toggles the cursor. It takes one argument that toggles the value between on (1) and off (0) using simplified directives:

```
            PUBLIC CURSW
            .MODEL small, C
            .CODE
CURSW PROC switch:word

      MOV         AX,SWITCH           ; get flag value
      XOR         AX,AX               ; test zero / nonzero
                  :
                  :
            // routine to test the switch and accordingly
```

```
switch off or switch on the cursor //
            :
            :
CURSW ENDP
            END
```

In a similar manner the variables can be passed in C as pointers also. Values can be returned to C either by changing the variable values in the C data segment or by returning the value in the registers as given earlier.

4.5 INTERRUPTS

Interrupts are signals that cause the central processing unit to suspend the currently executing program and transfer to a special program called an interrupt handler. The interrupt handler determines the cause of the interrupt, services the interrupt, and finally returns the control to the point of interrupt. Interrupts are caused by events external or internal to the CPU that require immediate attention. Some external events that cause interrupts are:

- Completion of an I/O process
- Detection of a hardware failure

An 8086 interrupt can occur because of the following reasons:

1. Hardware interrupts, caused by some external hardware device.
2. Software interrupts, which can be invoked with the help of INT instruction.
3. Conditional interrupts, which are mainly caused due to some error condition generated in 8086 by the execution of an instruction.

When an interrupt can be serviced by a procedure, it is called as the Interrupt Service Routine (ISR). The *starting addresses* of the interrupt service routines are present in the first 1K addresses of the memory (Please refer to Unit 2 of this block). This table is called the interrupt vector table.

How can we write an Interrupt Servicing Routine? The following are the basic but rigid sequence of steps:

1. Save the system context (registers, flags etc. that will be modified by the ISR).
2. Disable the interrupts that may cause interference if allowed to occur during this ISR's processing
3. Enable those interrupts that may still be allowed to occur during this ISR processing.
4. Determine the cause of the interrupt.
5. Take the appropriate action for the interrupt such as – receive and store data from the serial port, set a flag to indicate the completion of the disk sector transfer, etc.
6. Restore the system context.
7. Re-enable any interrupt levels that were blocked during this ISR execution.
8. Resume the execution of the process that was interrupted on occurrence of the interrupt.

MS-DOS provides you facilities that enable you to install well-behaved interrupt handlers such that they will not interfere with the operating system function or other interrupt handlers. These functions are:

Function	Action
Int 21h function 25h	Set interrupt vector
Int 21h function 35h	Get interrupt vector
Int 21h function 31h	Terminate and stay residents

Here are a few rules that must be kept in mind while writing down your own Interrupt Service Routines:

1. Use Int 21h, function 35h to get the required IVT entry from the IVT. Save this entry, for later use.
2. Use Int 21h, function 25h to modify the IVT.
3. If your program is not going to stay resident, save the contents of the IVT, and later restore them when your program exits.
4. If your program is going to stay resident, use one of the terminate and stay resident functions, to reserve proper amount of memory for your handler.

Let us now write an interrupt routine to handle "division by zero". This file can be loaded like a COM file, but makes itself permanently resident, till the system is running.

This ISR is divided into two major sections: the initialisation and the interrupt handler. The initialisation procedure (INIT) is executed only once, when the program is executed from the DOS level. INIT takes over the type zero interrupt vector, it also prints a sign-on message, and then performs a terminate and "stay resident exit" to MS-DOS. This special exit reserves the memory occupied by the program, so that it is not overwritten by subsequent application programs. The interrupt handler (ZDIV) receives control when a divide-by-zero interrupt occurs.

```
CR          EQU   ODH            ; ASCII carriage return
LF          EQU   0Ah            ; ASCII line feed
BEEP        EQU   07h            ; ASCII beep code
BACKSP      EQU   08h            ; ASCII backspace code

CSEG SEGMENT PARA PUBLIC  'CODE'
        ORG 100h
        ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG

INIT PROC NEAR
        MOV    DX,OFFSET ZDIV          ; reset interrupt 0 vector
                                       ; to address of new
                                       ; handler using function 25h, interrupt
        MOV    AX, 2500h               ; 0 handles divide-by-zero
        INT    21h
        MOV    AH,09                   ; print identification message
        INT    21h
                                       ; DX assigns paragraphs of memory
                                       ; to reserve
        MOV    DX,((OFFSET PGM_LEN + 15)/16) + 10h
        MOV    AX,3100h                ; exit and stay resident
        INT    21h                     ; with a return code = 0
INIT    ENDP

ZDIV PROC FAR                          ; this is the zero-divide
                                       ; hardware interrupt handler.
        STI                            ; enable interrupts.
        PUSH   AX                      ; save general registers
        PUSH   BX
        PUSH   CX
        PUSH   DX
        PUSH   SI
        PUSH   DI
        PUSH   BP
        PUSH   DS
        PUSH   ES
```

```
        MOV     AX,CS
        MOV     DS,AX
        MOV     DX,OFFSET WARN      ; Print warning "divide by
        MOV     AH, 9               ; zero "and" continue or
        INT     21h                 ; quit?"

ZDIV1:  MOV     AH,1                ; read keyboard
        INT     21h
        CMP     AL, 'C'             ; is it 'C' or 'Q'?
        JE      ZDIV3               ; jump it is a 'C'.
        CMP     AL, 'Q'

        JE      ZDIV2               ; jump it's a'Q'
        MOV     DX, OFFSET BAD      ; illegal entry, send a
        MOV     AH,9                ; beep, erase the bad char
        INT     21h                 ; and try again
        JMP     ZDIV1

ZDIV2:  MOV     AX, 4CFFh           ; user wants to abort the
        INT     21h                 ; program, return with
                                    ; return code = 255
ZDIV3:  MOV     DX,OFFSET CRLF      ; user wishes to continue
        MOV     AH,9                ; send CRLF
        INT     21h
        POP     ES                  ; restore general registers
        POP     DS                  ; and resume execution
        POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        IRET
ZDIV    ENDP

SIGNON          DB          CR, LF, 'Divide by zero interrupt'
                DB          'Handler Installed'
                DB          CRLF,'$'
WARN            DB          CR, LF, 'Divide by zero detected:'
                DB          CR, LF 'Quit or Continue (C/Q) ?'
                DB          '$'
BAD             DB          BEEP, BACKSP, '',BACKSP,'$'
CRLF            DB          CR,LF,$'
PGM_LEN         EQU $-INIT
CSEG    ENDS
                END
```

4.6 DEVICE DRIVERS IN ASSEMBLY

Device drivers are special programs installed by the config.sys file to control installable devices. Thus, personal computers can be expanded at some future time by the installation of new devices.

The device driver is .com file organized in 3 parts.

1) The leader
2) The strategy procedure

3) The interrupt procedure

The driver has either .sys or .exe extension and is originated at offset address 0000h.

The Header

The header contains information that allows DOS to identify the driver. It also contains pointers that allow it to chain to other drivers loaded into the system.

The header section of a device driver is 18 bytes in length and contains pointers and the name of the driver.

Following structure of the header:

```
CHAIN DD -1              : link to next driver
ATTR DW 0                : driver attribute
STRT DW START            : address of strategy
INTER DW INT             : address if interrupt
DNAME DB 'MYDRIVER'  : driver name.
```

The first double word contains a –1 that informs DOS this is the last driver in the chain. If additional drivers are added DOS inserts a chain address in this double word as the segment and offset address. The chain address points to the next driver in the chain. This allows additional drivers installed at any time.

The attribute word indicates the type of headers included for the driver and the type of device the driver installs. It also indicates whether the driver control a character driver or a block device.

The Strategy Procedure

The strategy procedure is called when loaded into memory by DOS or whenever the controlled device request service. The main purpose of the strategy is to save the request header and its address for use by the interrupt procedure.

The request header is used by DOS to communicate commands and other informations to the interrupt procedure in the device driver

The request header contains the length of the request header as its first byte. This is necessary because the length of the request header varies from command to command. The return status word communicate information back to DOS from the device driver.

The initialise driver command (00H) is always executed when DOS initialises the device driver. The initialisation commands pass message to the video display indicating that the driver is loaded into the system and returns to DOS the amount of memory needed by the driver. You may only use DOS INT 21H functions 00H. You can get more details on strategy from the further readings.

The Interrupt Procedure

The interrupt procedure uses the request header to determine the function requested by DOS. It also performs all functions for the device driver. The interrupt procedures must respond to at least the initialised driver command (00H) and any other commands required to control the device operated by the device driver. You must refer to the further readings for more details and examples of device drivers.

☛ Check Your Progress 3

State True or False

		T	F
(a)	Assembly language routines cannot be interfaced with BASIC programs.		
(b)	The key issue in interfacing is the selection of proper parameter passing method.		
(c)	The value of arguments to be passed are pushed in the stack in reverse order.		
(d)	AX, BX, CX or DX registers need not be saved in interfacing of assembly programs with high level language programs.		
(e)	Hardware interrupts can be invoked with the help of INT function.		

2. What are the sequences of steps in an interrupt service routine?

....

....

....

4.7 SUMMARY

In the above module, we studied some programming techniques, starting from arrays, to interrupts.

Arrays can be of byte type or word type, but the addressing of the arrays is always done with respect to bytes. For a word array, the address will be incremented by two for the next access.

As the programs become larger and larger, it becomes necessary to divide them into smaller modules called procedures. The procedures can be NEAR or FAR depending upon where they are being defined and from where they are being called. The parameters to the procedures can be passed through registers, or through memory or stack. Passing parameters in registers is easier, but limits the total number of variables that can be passed. In memory locations it is straight forward, but limits the use of the procedure. Passing parameters through stack is most complex out of all, but is a standard way to do it. Even when the assembly language programs are interfaced to high level languages, the parameters are passed on stack.

Interrupt Service Routines are used to service the interrupts that could have arisen because of some exceptional condition. The interrupt service routines can be modified- by rewriting them, and overwriting their entry in the interrupt vector table.

This completes the discussion on microprocessors and assembly language programming. The above programming was done for 8086 microprocessor, but can be tried on 80286 or 80386 processors also, with some modification in the assembler directives. The assembler used here is MASM, Microsoft assembler. The assembly language instructions remain the same for all assemblers, though the directives vary from one assembler to another. For further details on the assembler, you can refer to their respective manuals. You must refer to further readings for topics such as Interrupts, device drivers, procedures etc.

4.8 SOLUTIONS/ ANSWERS

Check Your Progress 1

1. We will give you an algorithm using XLAT instruction. Please code and run the program yourself.

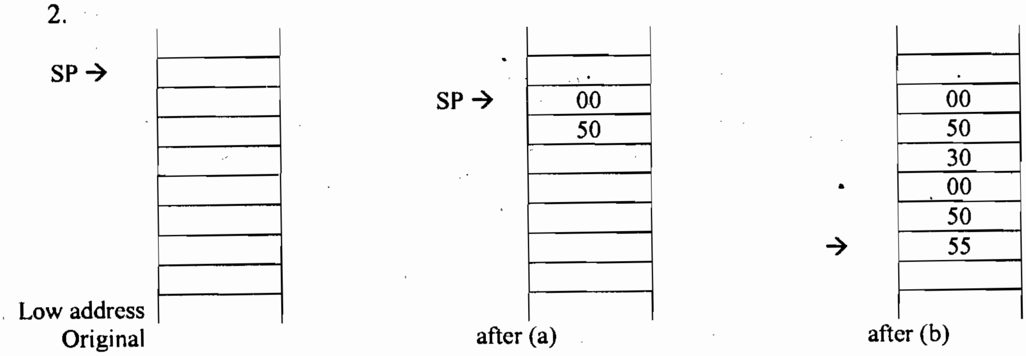
   - Take a sentence in upper case for example 'TO BE CONVERTED TO LOWER CASE' create a table for lower case elements.
   - Check that the present ASCII character is an alphabet in upper case. It implies that ASCII character should be greater than 40h and less than 58h.
   - If it is upper case then subtract 41h from the ASCII value of the character. Put the resultant in AL register.
   - Set BX register to the offset of lower case table.
   - Use XLAT instruction to get the required lower case value.
   - Store the results in another string.

2. (a) False (b) False (c) True

Check Your Progress 2

1. (a) True (b) False (c) False (d) True (e) False (f) True (g) True (h) False (i) False (j) True.

2.



(c) The return for FIRST can occur only after return of SECOND. Therefore, the stack will be back in original state.

Check Your Progress 3

1. (a) False (b) False (c) True (d) True (e) False

2.
   - Save the system context
   - Block any interrupt, which may cause interference
   - Enable allowable interrupts
   - Determine the cause of interrupt
   - Take appropriate action
   - Restore system context
   - Enable interrupts which were blocked in Step 2