# Experiment 8

# 8 Subroutine Handling Instructions and Macros

**Introduction**

In this experiment you will be introduced to subroutines and how to call them. You will verify the exchange of data between a main program and a subroutine in the 8086 environment. You will also use Macros, and as applications you will deal to a useful data representation: look-up tables. .

You will need some of the programs developed in previous experiments to rewrite them in a more structured way.

**Objectives**

1- Stack
2- PUSH and POP instructions
3- Procedures and Procedure Calls
4- Parameter Passing through Memory, Registers and the Stack
5- Macros
6- Application: Real-time clock reading,

## *8.1 Procedures*

### 8.1.1 The Stack

The stack is a LIFO data structure in memory used to facilitate subroutine handling. The SS register contains the Stack Segment number where the stack is stored. The "**.STACK**" directive instructs the assembler to reserve a stack of a desired size. If the ".STACK" directive is missing from a program, the assembler issues the warning:

**LINK: Warning L4021: no stack segment**.

The stack always starts at a high address and grows towards the beginning of the stack segment at a lower address. The microprocessor stores data on the stack as needed, and uses the SP register to point to the last item stored on the stack. The stack size dynamically changes as data is stored or retrieved from the stack.

### 8.1.2 Stack Handling Instructions

There exist a number of data transfer instructions that are used with the stack. These are summarized in the following.

### 8.1.2.1 PUSH and POP

There exist two direct stack handling instructions (Table 8.1): PUSH and POP. The PUSH instruction is used to store the content of a 16-bit register, or memory location, on the stack. It first decreases the content of SP by two and then stores the data into the two bytes on the top of the stack. The high order byte of the data goes to the high addressed byte in the stack. The POP instruction retrieves a word from the stack and then increases SP by two.

**Example**:

The directive

**.STACK 100H**

instructs the assembler to reserve a stack of size 256 bytes and initializes the Stack Pointer register (SP) to 100H.

| Instruction | Example | Meaning |
|---|---|---|
| PUSH | PUSH AX | SP ← SP –2<br>[SP-1] ← AH<br>[SP-2] ← AL |
| POP | POP NUM1 | [UM+1] ← [SP+1]<br>[NUM1] ← [SP]<br>SP ← SP + 2 |

**Table 8. 1**: Basic the Satck Handling Instructions

| | | |
|---|---|---|
| PUSHF | PUSHF | SP ← SP –2<br>[SP-1] ← MSB(FR)<br>[SP-2] ← LSB(FR) |
| POPF | POPF | LSB(FR) ←[SP]<br>MSB(FR) ←[SP+1]<br>SP ← SP + 2 |

**Note:** FR = Flag Register

**Table 8. 2**: Flag Register Handling Instructions

### 8.1.2.2 PUSHF and POPF

Two other instructions are used to manipulate the flag register (FR) only: PUSHF and POPF (Table 8.1). The PUSHF instruction is similar to the PUSH instruction, except that the PUSHF is used to push the contents of the flag register onto the stack. The POP and POPF instructions have a reverse action of the PUSH and PUSHF, respectively. The POPF has the same effect, except that the word retrieved is saved to the flag register. These two instructions are used to access some of the bits of the flag register that are not accessible by appropriate instructions. Note that not all flags are accessible to the programmer, but only the ones shown in **Error! Reference source not found.**2 which are directly accessible by appropriate instructions.

The remaining flags can be accessed by POPF and PUSF instructions. Once saved in the AX register, those flags can be set, reset or flipped using bitwise instructions. The programmer needs to know their exact locations to be able to access them (Figure 8.1).

| Flag | Instruction | Effect |
|---|---|---|
| **Carry Flag** | **STC** | Set Carry flag |
| | **CLC** | Clear Carry flag |
| **Direction Flag** | **STD** | Set Direction flag |
| | **CLD** | Clear Direction flag |
| **Interrupt Flag** | **STI** | Set Interrupt flag |
| | **CLI** | Clear Interrupt flag |

**Table 8. 3**: Flags Directly Accessible

The following figure shows the positions of the flags that require some manipulations of the AX Register to be accessed:

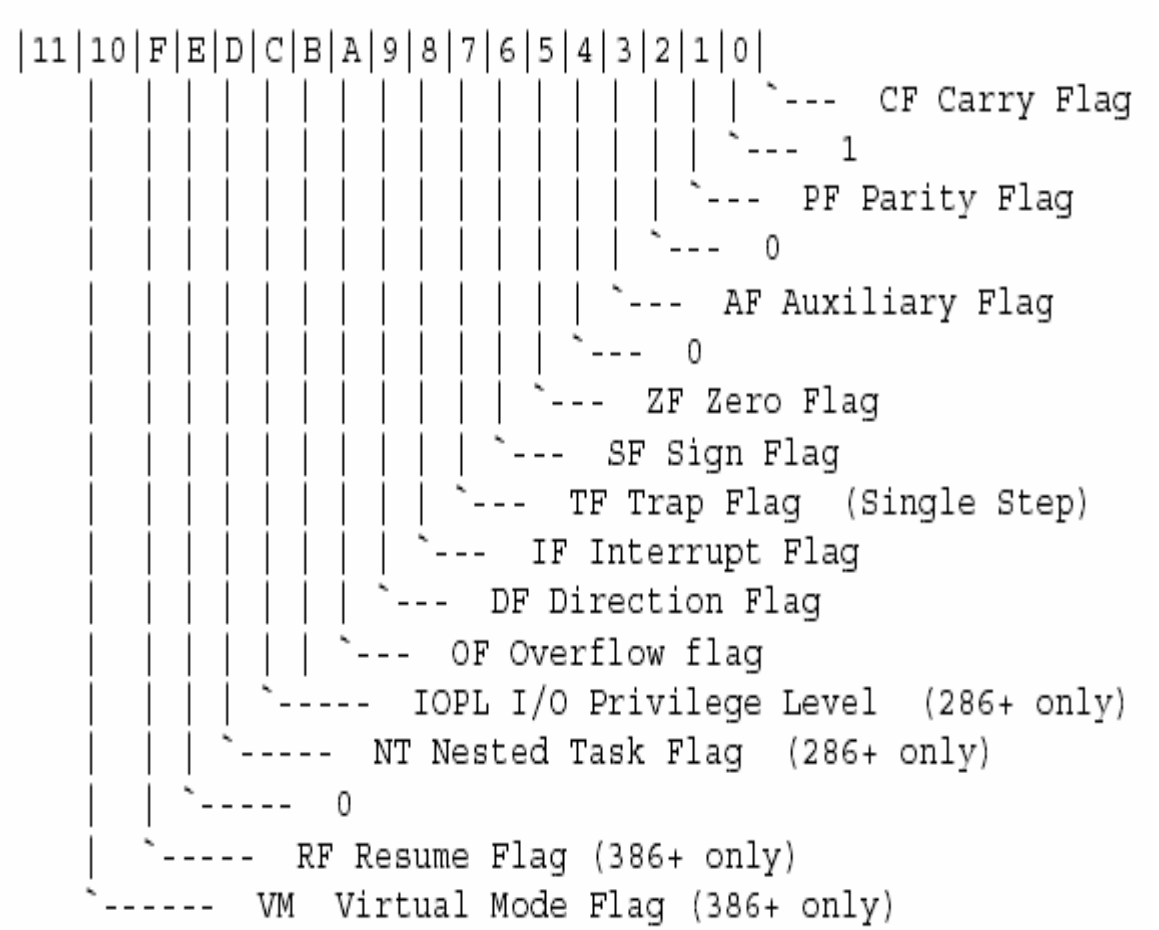


**Figure 8.1**: Flag register

#### 8.1.2.3 PUSHA and POPA

These are used to push all general registers as shown in Table 8.3.

| | | |
|---|---|---|
| PUSHA | PUSHA | Push AX CX DX BX original SP BP SI and DI |
| PUSHAD | PUSHAD | Push EAX ECX EDX EBX original ESP EBP ESI and EDI |
| POPA | POPA | Push AX CX DX BX original SP BP SI and DI |
| POPAD | POPAD | Push EAX ECX EDX EBX original ESP EBP ESI and EDI |

**Table 8. 4**: Flags Directly Accessible

### 8.1.3 Subroutine Calls

A procedure is a reusable section of the software that is stored in memory once, but used as often as necessary. The CALL instruction links to the procedure and the RET (return) instruction returns from the procedure. The Stack stores the return address whenever a procedure is called during the execution of a program. The CALL instruction pushes the address of the instruction following the CALL (return address) onto the stack. The RET instruction removes an address from the stack, so the program returns to the instruction following the CALL.

With the Assembler (MASM) there are specific ways for writing, and storing, procedures. A procedure begins with the PROC directive and ends with the ENDP directive. Each directive appears with the name of the procedure. The PROC directive is followed by the type of the procedure: NEAR (intra-segment) or FAR (inter-segment).

In MASM version 6.X, a NEAR or FAR procedure can be followed by the USES statement. The USES statement allows any number of registers to be automatically pushed onto the stack and popped from the stack within the procedure.

Procedures that are to be used by all software (*global*) should be written as FAR procedures. Procedures that are used by a given task (*local*) are normally defined as NEAR procedures.

### 8.1.4 The CALL Instruction

The CALL instruction transfers the flow of the program to the procedure. The CALL instruction differs from the jump instruction in the sense that a CALL saves a return address on the stack. The RET instruction return control to the instruction that immediately follows the CALL. There exist two types of calls: FAR and NEAR, and two types of addressing modes used with calls, Register and Indirect Memory modes.

#### 8.1.4.1 Near CALL

A near CALL is three bytes long, with the first byte containing the opcode, and the two remaining bytes containing the displacement or distance of ±32 K. When a NEAR CALL executes, it pushes the offset address of the next instruction on the stack. The offset address of the next instruction appears in the IP register. After saving this address, it then adds the displacement from bytes 2 and 3 to the IP to transfer control to the procedure. A variation of NEAR CALL exists, CALLN, but should be avoided.

#### 8.1.4.2 Far CALL

The FAR CALL can call a procedure anywhere in the system memory. It is a five-byte instruction that contains an opcode followed by the next value for the IP and CS registers. Bytes 2 and 3 contain the new contents of IP, while bytes 4 and 5 contain the new contents for CS. The FAR CALL instruction places the contents of both IP and CS on the stack before jumping to the address indicated by bytes 2 to 5 of the instruction. This allows a call to a procedure anywhere in memory and return from that procedure. A variant of the FAR CALL is CALLF but should be avoided.

### 8.1.5 Calls with Register Operand

CALLs can contain a register operand. An example is CALL BX, in which the content of IP is pushed into the stack, and a jump is made to the offset address located in register BX, in the current code segment. This type of CALL uses a 16-bit offset address stored in any 16-bit register, except the segment registers.

Program 8.1 illustrates the use of the CALL register instruction to call a procedure that begins at offset address DISP. The offset address DISP is placed into the BX register, then the CALL BX instruction calls the procedure beginning at address DISP. This program displays “OK” on the monitor screen.

### 8.1.6 CALL’s with Indirect Memory Address

A CALL with an indirect memory address is useful when different subroutines need to be chosen in a program. This selection process is often keyed with a number that addresses a CALL address in a lookup table.

Program 8.2 shows three separate subroutines referenced by Number 1,2 and 3 as read from the keyboard. The calling sequence adjusts the value of AL and extends it to a 16-bit number before adding it to the location of the lookup table. This references one of the three subroutines using the CALL TABLE[BX] instruction. When this program executes, the letter A is displayed when a 1 is typed, B if 2 and C if 3 is typed.

The CALL instruction can also reference far pointers if the data in the table are defined as double-word data with the DD directive, using the CALL FAR PTR[SI] or CALL

TABLE[SI] instructions. These instructions retrieve a 32-bit address from the data segment memory location addressed by SI and use it as the address of a far procedure.

## *8.2 Parameter Passing*

To pass data (parameters) between the main program and the routines, data may be left in the general-purpose registers. This method has the disadvantage of changing the contents of the registers every time the subroutine is called. A more elegant way is to exchange data through the stack, or through memory. The data to be passed to a subroutine is saved in the memory before calling the subroutine. All the registers that need to be saved, and are used by the subroutine, should also be saved and retrieved afterwards.

| **Instruction** | **Example** | **Effect** |
|---|---|---|
| **CALL** | CALL SQRT | [SP-1] ← 34<br>[SP-2] ← 5B<br>SP ← SP-2<br>IP ← 34A0 |
| **RET** | RET | LSB(IP) ← [SP]<br>MSB(IP) ← [SP+1]<br>SP ← SP + 2 |

**Note**: Assuming SQRT is a Near Procedure, starting at CS:34A0H, and the instruction CALL is at CS:345BH.

**Table 8. 5:** Summary of the Subroutine Handling Instructions

## *8.3 Macros*

Macro sequences relieve the programmer from retyping the same instructions. They allow you create your own pseudo language for instruction sequences that often appear in programming. A macro sequence starts by the MACRO directive and ends by an ENDM directive. Associated with MACRO is the name of the macro and any parameters that are carried with the macro to the instructions between MACRO and ENDM statements. Program 8.3 contains two macros. A MACRO is declared and used as shown in the following example

```
DISPLAY MACRO STRING
      MOV DX,OFFSET STRING
      MOV AH,09H
      INT 21H
ENDM

;If “Message” is the string to be displayed, the


; Macro is called as follows:
DISPLAY MESSAGE
```

Macros can be saved in a separate file, to which a name such as “MACRO.INC” can be given. This file can then be used as a library, and therefore can be included in the program using the directive INCLUDE, in the following manner:

**INCLUDE MACRO.INC**

Provided that both, the program and the macro library are in the same directory. Alternatively the path has to be specified as follows:

**INCLUDE Path\MACRO.INC**

## *8.4 Labels local to a Macro:*

When a MACRO contains labels, and the Macro is used more than once in a program, which is usually the case, the assembler gives the following error: Label referenced more than once. To avoid such an error, these labels should be made local to the MACRO, this is done using the following:

```
DISPLAY MACRO STRING
Local Label1
      …

Label1:…
      …
ENDM
```

**Reading the System Date: Function 2AH, INT 21H:**

Function 2AH of Interrupt INT 21H is used to read the system date. It returns the Day of the week in AL register, the year in CX register, the month in DH register and the day of the month in DL register. Note that as indicated in Table 8. 2, the returned values are in hexadecimal format, which, in order to be displayed, need to be converted to decimal, as indicated in experiment 5.

| **Function** | **Effect**: Read system date | | |
|---|---|---|---|
| | **ENTRY** | **EXIT** | |
| **2AH** | AH = 2AH | AL = Day of the week | CX = Year (1980-2099) |
| | | DH = Month | DL = Day of the month |
| **Note:** The day of the week is encoded as Sunday = 00 through Saturday = 06. The year is a binary number equal to 1980 through 2099. | | | |

**Table 8. 3:** Read Time and Date Function: 2AH, INT 21H

## *8.5 Lab Work*

**Pre Lab Work**:

1. Write, assemble, link and run program 8.1 and 8.2. Try to understand how the different routines are written and how they are called. See also how the different procedures pass parameters between them.
2. Write, assemble, link and run program 8.3. See how Macros are used.
3. Rewrite program 6.1, from Experiment 6, using Procedures and Macros. Call it program 8.4.
4. Bring your work to the lab.

**Lab Work:**

1- Show programs 8.3 and 8.4 to your lab instructor.
2- Modify program 6.3, from experiment 6, using Procedures and Macros. Call it program 8.5.
3- Program 8.4 reads a string and encrypts it. Complete the program and use Macros and Procedures.
4- Modify Program 8.4, so that it reads an encrypted string and converts it back to the original one. Write this program using procedures and Macros. Call it program 8.6.

**Lab Assignment:**

DOS Function 2CH reads the system time, and works as described below:

```
MOV AH, 2CH
INT 21H
; and returns (in binary) the time as follows:
CH:   hours (0-23);
CL:   minutes (0-59);
DH:  seconds (0-59); and
DL:   hundredths of a second.
```

Use program 8.3, and the above function, to develop a program that reads the date, and displays it in the following format:

Today is: Sunday 24/October/1999, The Time is: 12:02:32

Make the program loop for a large number of times, so that you can see the time changing.

**TITLE "Program 8.1"**

```
;   a program that display OK on the monitor screen using procedure DISP

.MODEL TINY                     ; select TINY model
.CODE                       ; indicate start of CODE segment
.STARTUP                    ; indicate start of program

        MOV  BX, OFFSET DISP    ; address DISP with BX
        MOV  DL, 'O'            ; display 'O'
        CALL BX
        MOV  DL, 'K'            ; display 'K'
        CALL BX
.EXIT                           ; exit to DOS
;
;  a procedure that displays the ASCII contents of DL on the monitor screen.
;  ************************************************
DISP   PROC NEAR
        MOV  AH, 02             ; select function 02H
        INT 21H                 ; execute DOS function
RET                             ; return from procedure
DISP   ENDP

END                             ; end of program
```

---

**TITLE "Program 8.2"**

```
;   program that uses a CALL lookup table to access one of three different procedures:
;   ONE,  TWO,  or THREE.

.MODEL SMALL                    ;  select SMALL model

.DATA                                   ;  indicate start of DATA segment
        TABLE       DW    ONE           ;  define lookup table
                    DW    TWO
                    DW    THREE

.CODE                           ;  indicate start of CODE segment
 ONE   PROC NEAR
        MOV  AH, 2              ;  display a letter A
        MOV  DL, 'A'
        INT    21H
        RET
ONE    ENDP

TWO    PROC NEAR
        MOV  AH, 2              ;  display letter B
        MOV  DL, 'B'
        INT    21H
        RET
TWO    ENDP
```

```
THREE       PROC NEAR
        MOV  AH, 2          ; display the letter C
        MOV  DL, 'C'
        INT  21H
        RET
THREE       ENDP

;   Start of Main Program
.STARTUP
TOP:
        MOV  AH, 1          ; read key into AL
        INT  21H
        SUB  AL, 31H              ; convert from ASCII to 0, 1, or 2
        JB   TOP            ; if below 0
        CMP  AL, 2
        JA   TOP            ; if above 2
        MOV  AH, 0          ; form lookup address
        MOV  BX, AX
        ADD  BX, BX
        CALL TABLE [BX]  ; call procedure ONE,  TWO,  or THREE
.EXIT                       ; exit to DOS
END                         ; end of file
```

---

**TITLE “Program 8.3”**

```
; This program uses the function read time and displays the current day

.MODEL SMALL                                ;  select SMALL model
.STACK              100
.DATA
                CRLF                DB      0DH,0AH,'$'
                PROMPT1     DB      'Today is : ','$'

                DAY     DW      D0,D1,D2,D3,D4,D5,D6
                D0      DB      ‘SUNDAY’,’$’
                D1      DB      ‘MONDAY’,’$’
                D2      DB      ‘TUESDAY’,’$’
                D3      DB      ‘WEDNESDAY’,’$’
                D4      DB      ‘THURSDAY’,’$’
                D5      DB      ‘FRIDAY’,’$’
                D6      DB      ‘SATURDAY’,’$’

.CODE
.STARTUP
                ; Display Prompt1
                MOV AH, 2AH                     ; GET SYSTEM DATE
                INT 21H
                MOV SI, OFFSET DAY
                MOV AH, 00
                ADD AX, AX
```

```
                ADD SI, AX
                MOV DX, [SI]
                MOV AH, 09H
                INT 21H

                LEA DX, CRLF                    ; MOVE CURSOR TO NEXT LINE
                MOV AH, 09H
                INT 21H
.EXIT
END
```

---

**TITLE "Program 8.4"**

```
; This program reads a string of 200 characters maximum and encrypts ; it.
.MODEL SMALL
.STACK 100
.DATA
      CRLF    DB 0DH,0AH,'$'
      PROMPT1 DB 'Enter a string : ','$'
      STRING  DB 100 DUP(?)
      CODED   DB 100 DUP(?)
      UTAB    DB 'MNBVCXZLKJHGFDSAPOIUYTREWQV'
      LTAB    DB 'bgtnhymjukilopvfrcdexswzaq'

.CODE
.STARTUP

         ; DISPLAY PROMPT1
    ; Read a string from the keyboard, save it in the array STRING

         ; Scan the string STRING, and do the following:
         ; if character is an upper case letter
         ; that is:
         ; if AL >= 'A' and AL =< 'Z'

           MOV BX, OFFSET LTAB
      SUB AL, 41H
      XLAT
    ; Save the character in AL in the array CODED.
    ; if character is a lower case letter
    ; i.e.
    ; if AL >= 'a' and AL =< 'z'
      MOV BX, OFFSET UTAB
      SUB AL, 61H
      XLAT
    ; Save the character in AL in the array CODED.
    ; MOVE CURSOR TO NEXT LINE
         ; Display the array CODED.
         ; exit to DOS
END
```