Unit 4

4.1 LINKING AND RELOCATION

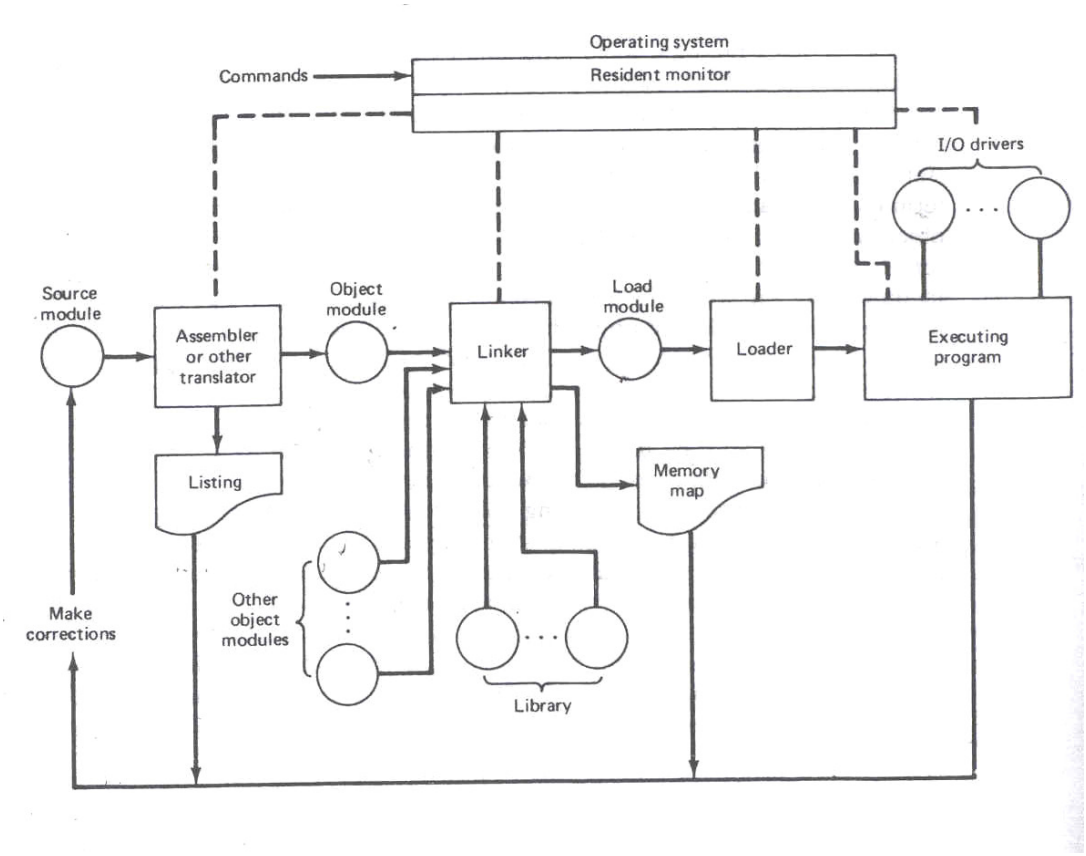


Figure 4.1 Creation and execution of a program

The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM.

The linker program is invoked using the following options.

C> LINK

or

C>LINK MS.OBJ

The .OBJ extension is a must for a file to be accepted by the LINK as a valid object file. The first object may generate a display asking for the object file, list file and libraries as inputs and an expected name of the .EXE file to be generated. The output of the link program is an executable file with the entered filename and .EXE extension. This executable filename can further be entered at the DOS prompt to execute the file.

In the advanced version of the MASM, the complete procedure of assembling and linking is combined under a single menu invokable compile function. The recent versions of MASM have much more sophisticated and user-friendly facilities and options. A linker links the machine codes with the other required assembled codes. Linking is necessary because of the number of codes to be linked for the final binary file.

The linked file in binary for ***run*** on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory. The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant.

The DOS linking program links the different object modules of a source program and function library routines to generate an integrated executable code of the source program. The main input to the linker is the .OBJ file that contains the object modules of the source programs. Other supporting information may be obtained from the files generated by the MASM.

The linked file in binary for ***run*** on a computer is commonly known as executable file or simply '.exe.' file. After linking, there has to be re-allocation of the sequences of placing the codes before actually placement of the codes in the memory. The loader program performs the task of reallocating the codes after finding the physical RAM addresses available at a given instant.

The ***loader*** is a part of the operating system and places codes into the memory after reading the '.exe' file. This step is necessary because the available memory addresses may not start from 0x0000, and binary codes have to be loaded at the different addresses during the run. The loader finds the appropriate start address.

In a computer, the loader is used and it loads into a section of RAM the program that is ready to run. A program called *locator* reallocates the linked file and creates a file for permanent location of codes in a standard format.

4.1.1 Segment combination

In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker. Segments with same name are joined together by using the modifiers attached to the SEGMENT directives. SEGMENT directive may have the form

Segment name SEGMENT Combination-type

where the combine-type indicates how the segment is to be located within the load module. Segments that have different names cannot be combined and segments with the same name but no combine-type will cause a linker error. The possible combine-types are:

PUBLIC – If the segments in different modules have the same name and combine-type PUBLIC, then they are concatenated into a single element in the load module. The ordering in the concatenation is specified by the linker command.

COMMON – If the segments in different object modules have the same name and the combine-type is COMMON, then they are overlaid so that they have the same starting address. The length of the common segment is that of the longest segment being overlaid.

STACK – If segments in different object modules have the same name and the combine-type STACK, then they become one segment whose length is the sum of the lengths of the individually specified segments. In effect, they are combined to form one large stack

AT – The AT combine-type is followed by an expression that evaluates to a constant which is to be the segment address. It allows the user to specify the exact location of the segment in memory.

MEMORY – This combine-type causes the segment to be placed at the last of the load module. If more than one segment with the MEMORY combine-type is being linked, only the first one will be treated as having the MEMORY combine type; the others will be overlaid as if they had COMMON combine-type.

Source module 1

```
DATA        SEGMENT     COMMON
DATA                ENDS
CODE                SEGMENT      PUBLIC
CODE                ENDS
```

Source module 2

```
DATA        SEGMENT     COMMON
                .
                .
DATA        ENDS
CODE        SEGMENT      PUBLIC
                .
                .
CODE        ENDS
```

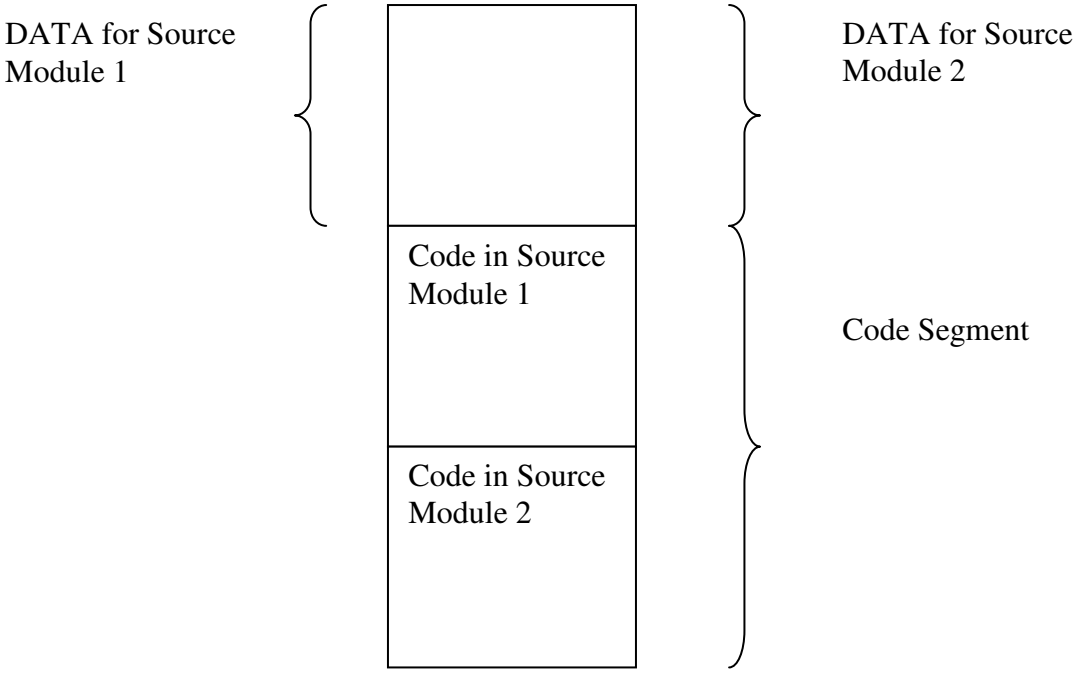


Figure 4.2 Segment combinations resulting from the PUBLIC and COMMON combination types

Source module 1

```
STACK_SEG   SEGMENT STACK
        DW     20 DUP (?)
        TOP-OF_STACK LABEL WORD
STACK_SEG  ENDS
        END
```

Source module 2

```
        STACK_SEG          SEGMENT     STACK
         DW     30 DUP (?)
STACK_SEG          ENDS
                .
                .
END
```

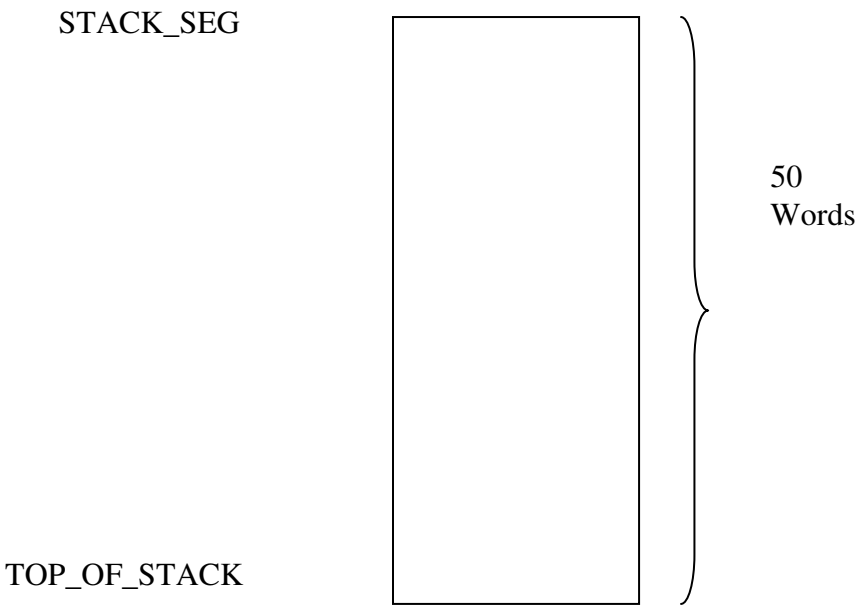


Figure 4.3 Formation of a stack from two segments

4.1.2 Access to External Identifiers

If an identifier is defined in an object module, then it is said to be a *local* (or *internal*) *identifier* relative to the module. If it is not defined in the module but is defined in one of the other modules being linked, then it is referred to as an *external* (or *global*) *identifier* relative to the module.In order to permit other object modules to reference some of the identifiers in a given module, the given module must include a list of the identifiers to which it will allow access. Therefore, each module in multi-module programs may contain two lists, one containing the external identifiers that can be referred to by other modules.

Two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

EXTRN Identifier: Type…, Identifier: Type

and

PUBLIC Identifier,…, Identifier

where the identifiers are the variables and labels being declared or as being available to other modules.

The assembler must know the type of all external identifiers before it can generate the proper machine code, a type specifier must be associated with each identifier in an EXTRN statement.

For a variable the type may be BYTE, WORD, or DWORD and for a label it may be NEAR or FAR.

One of the primary tasks of the linker is to verify that every identifier appearing in an EXTRN statement is matched by one in a PUBLIC statement. If this is not the case, then there will be an undefined reference and a linker error will occur.

The offsets for the local identifier will be inserted by the assembler, but the offsets for the external identifiers and all segment addresses must be inserted by the linking process.The offsets associated with all external references can be assigned once all of the object modules have been found and their external symbol tables have been examined.

The assignment of the segment addresses is called *relocation* and is done after the linking process has determined exactly where each segment is to be put in memory.

4.2 STACKS

The stack is a block of memory that may be used for temporarily storing the contents of the registers inside the CPU. It is a top-down data structure whose elements are accessed using the stack pointer (SP) which gets decremented by two as we store a data word into the stack and gets incremented by two as we retrieve a data word from the stack back to the CPU register.

The process of storing the data in the stack is called ‘pushing into’ the stack and the reverse process of transferring the data back from the stack to the CPU register is known as ‘popping off’ the stack. The stack is essentially *Last-In-First-Out* (LIFO) data segment. This means that the data which is pushed into the stack last will be on top of stack and will be popped off the stack first. The stack pointer is a 16-bit register that contains the offset address of the memory location in the stack segment.

The stack segment, like any other segment, may have a memory block of a maximum of 64 Kbytes locations, and thus may overlap with any other segments. Stack Segment register (SS) contains the base address of the stack segment in the memory.

The Stack Segment register (SS) and Stack pointer register (SP) together address the stack-top as explained below:

SS ⟹ 5000H

SP ⟹ 2050H

Physical address of Stack-top = 5000H * 10H + 2050H

If the stack top points to a memory location 52050H, it means that the location 52050H is already occupied with the previously pushed data. The next 16 bit push operation will decrement the stack pointer by two, so that it will point to the new stack-top 5204EH and the decremented contents of SP will be 204EH . This location will now be occupied by the recently pushed data.

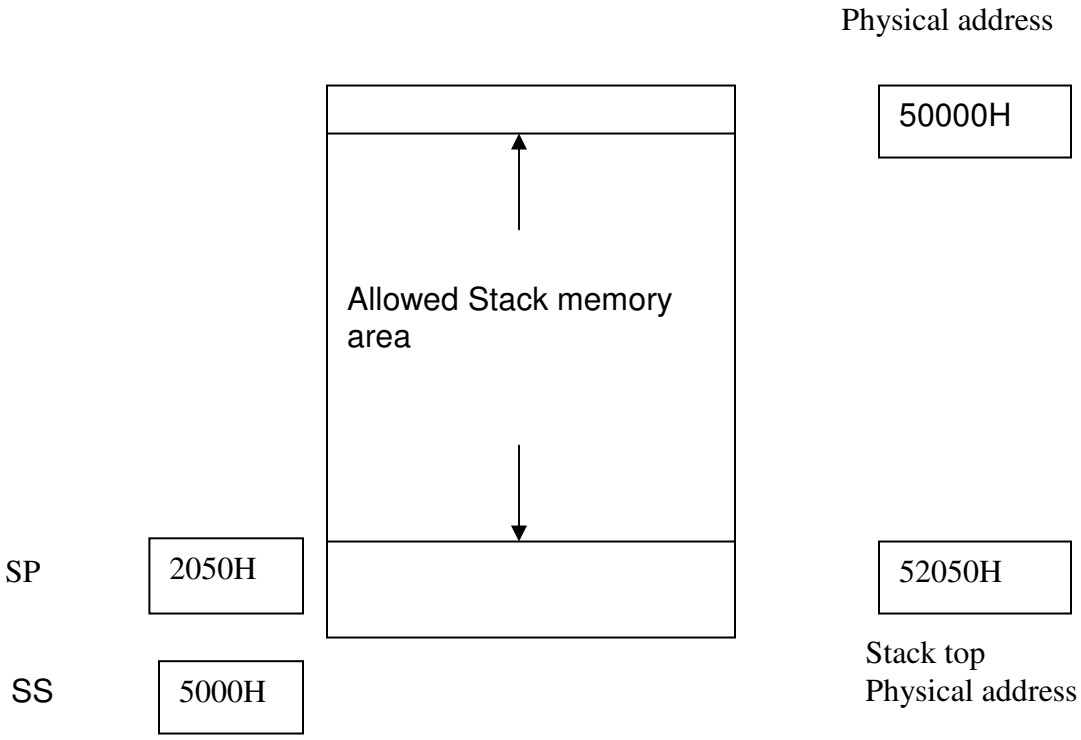


Fig. 4.4 Stack-top address calculation

Thus for a selected value of SS, the maximum value of SP=FFFFH and the segment can have maximum of 64K locations.

If the SP starts with an initial value of FFFFH, it will be decremented by two whenever a 16-bit data is pushed onto the stack. After successive push operations, when the stack pointer contains 0000H, any attempt to further push the data to the stack will result in stack overflow.

After a procedure is called using the CALL instruction, the IP is incremented to the next instruction. Then the contents of IP,CS and flag register are pushed automatically to the stack. The control is then transferred to the specified address in the CALL instruction i.e. starting address of the procedure. Then the procedure is executed.

4.2.1 Programming for Stack

Program 4.1

WAP to calculate squares of BCD numbers 0 to 9 and store them from 2000h offset onwards in the current data segment. The numbers and their squares are in the BCD format. Write a subroutine for the calculation of the square of a number.

```
ASSUME CS:CODE,DS:DATA,SS:STACK
DATA SEGMENT
ORG 2000H
SQUARES DB 0FH DUP (?)
DATA ENDS
STACK SEGMENT
STACKDATA DB 100H DUP (?)  ;Reserve 256 bytes for stack
STACK ENDS
CODE SEGMENT
START: MOV AX,DATA ;Initialize data segment
        MOV DS,AX
        MOV AX,STACK ;Initialize stack segment
        MOV SS,AX
       MOV SP,OFFSET STACKDATA ; Initialize stack pointer
       MOV CL,0AH          ; Initialize counter for numbers
MOV SI,OFFSET SQUARES; Pointer for array of squares
```

```
        MOV AL,00H          ; Start from zero
NEXTNUM: CALL SQUARE ; Calculate square procedure
         MOV BYTE PTR [SI],AH ; Store square in the array
         INC AL             ; Go to next number
         INC SI             ; Increment array pointer
         DCR CL             ; Decrement counter
         JNZ NEXTNUM        ; Stop if CL=0, else continue
         MOV AH,4CH
         INT 21H

PROCEDURE SQUARE NEAR ; Square is a near procedure
MOV BH,AL
MOV CH,AL
XOR AL,AL
PUSH BH
AGAIN: ADD AL,CH        ; Successively add CH to AL
       DAA              ; Get BCD equivalent
       DCR CH           ; Decrement multiplier register
       JNZ AGAIN
MOV AH,AL           ; Store the square of the number
POP BH
MOV AL,BH           ; Get back the number
RET
SQUARE ENDP
CODE ENDS
END START
```

Program 4.2

WAP to program change a sequence of Sixteen 2-byte numbers from ascending to descending order. The numbers are stored in the data segment. Store the new series at addresses starting from 6000H. Use LIFO property of stack.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
LIST DW 10H
ORG 6000H
RESULT DW 10H
COUNT EQU 10H
STACKDATA DB FFH DUP (?)
CODE SEGMENT
START: MOV AX,DATA ;Initialize data segment
        MOV DS,AX
        MOV SS,AX
        MOV SP,OFFSET LIST
        MOV CL,COUNT
        MOV BX, OFFSET RESULT+COUNT
NEXT:POP AX
      MOV DX,SP
      MOV SP,BX
PUSH AX
 MOV BX,SP
MOV SP,DX
DEC CL
JNZ NEXT
MOV AH,4CH
INT 21H
CODE ENDS
END START
```

4.3 PROCEDURES

A procedure is a set of code that can be branched to and returned from in such a way that the code is as if it were inserted at the point from which it is branched to. The branch to procedure is referred to as the *call,* and the corresponding branch back is known as the *return*. The return is always made to the instruction immediately following the call regardless of where the call is located.

4.3.1 Calls, Returns, and Procedure Definitions

The CALL instruction not only branches to the indicated address, but also pushes the return address onto the stack. The RET instruction simply pops the return address from the stack. The registers used by the procedure need to be stored before their contents are changed, and then restored just before their contents are changed, and then restored just before the procedure is excited.

A CALL may be direct or indirect and intrasegment or intersegment. If the CALL is intersegment , the return must be intersegment. Intersegment call must push both (IP) and (CS) onto the stack. The return must correspondingly pop two words from the stack. In the case of intrasegment call, only the contents of IP will be saved and retrieved when call and return instructions are used.

Procedures are used in the source code by placing a statement of the form at the beginning of the procedure

Procedure name PROC Attribute

and by terminating the procedure with a statement

Procedure name ENDP

The attribute that can be used will be either NEAR or FAR. If the attribute is NEAR, the RET instruction will only pop a word into the IP register, but if it is FAR, it will also pop a word into the CS register.

A procedure may be 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 could be NEAR provided that all calls are in the same code segment as the procedure. For the latter two cases the attribute must be FAR. If the procedure is given a FAR attribute, then all calls to it must be intersegment calls even if the call is from the same code segment. For the third case, the procedure name must be declared in EXTRN and PUBLIC statements.

4.3.2 Saving and Restoring Registers

When both the calling program and procedure share the same set of registers, it is necessary to save the registers when entering a procedure, and restore them before returning to the calling program.

```
MSK   PROC NEAR
      PUSH AX
      PUSH BX
      PUSH CX
      POP CX
      POP BX
      POP AX
      RET
MSK ENDP
```

4.3.3 Procedure Communication

There are two general types of procedures, those operate on the same set of data and those that may process a different set of data each time they are called. If a procedure is in the same source module as the calling program, then the procedure can refer to the variables directly.

When the procedure is in a separate source module it can still refer to the source module directly provided that the calling program contains the directive

```
PUBLIC ARY, COUNT, SUM
EXTRN ARY: WORD, COUNT: WORD, SUM: WORD
```

4.3.4 Recursive Procedures

When a procedure is called within another procedure it called recursive procedure.

To make sure that the procedure does not modify itself, each call must store its set of parameters, registers, and all temporary results in a different place in memory

Recursive procedure to compute the factorial

```
FACT1 PROC
      CMP BX,01H
      JZ LP
      PUSH BX
      DEC BX
      CALL FACT1
      POP BX
      MUL BX
      RET
LP:MOV AX,01H
   RET
```

4.4 INTERRUPTS AND INTERRUPT ROUTINES

4.4.1 Interrupt and its Need

The microprocessors allow normal program execution to be interrupted in order to carry out a specific task/work. The processor can be interrupted in the following ways

i) by an external signal generated by a peripheral,

ii) by an internal signal generated by a special instruction in the program,

iii) by an internal signal generated due to an exceptional condition which occurs while executing an instruction. (For example, in 8086 processor, divide by zero is an exceptional condition which initiates type 0 interrupt and such an interrupt is also called execution) .

In general, the process of interrupting the normal program execution to carry out a specific task/work is referred to as interrupt.

The interrupt is initiated by a signal generated by an external device or by a signal generated internal to the processor. When a microprocessor receives an interrupt signal it

stops executing current normal program, save the status (or content) of various registers (IP, CS and flag registers in case of 8086) in stack and then the processor executes a subroutine/procedure in order to perform the specific task/work requested by the interrupt. The subroutine/procedure that is executed in response to an interrupt is also called Interrupt Service Subroutine. (ISR). At the end of ISR, the stored status of registers in stack is restored to respective registers, and the processor resumes the normal program execution from the point {instruction) where it was interrupted.

The external interrupts are used to implement interrupt driven data transfer scheme. The interrupts generated by special instructions are called software interrupts and they are used to implement system services/calls (or monitor services/calls). The system/monitor services are procedures developed by system designer for various operations and stored in memory. The user can call these services through software interrupts. The interrupts generated by exceptional conditions are used to implement error conditions in the system.

4.4.2 Interrupt Driven Data Transfer Scheme

The interrupts are useful for efficient data transfer between processor and peripheral. When a peripheral is ready for data transfer, it interrupts the processor by sending an appropriate signal. Upon receiving an interrupt signal, the processor suspends the current program execution, save the status in stack and executes an ISR to perform the data transfer between the peripheral and processor. At the end of ISR the processor status is restored from stack and processor resume its normal program execution. This type of data transfer scheme is called interrupt driven data transfer scheme.

The data transfer between the processor and peripheral devices can be implemented either by polling technique or by interrupt method. In polling technique, the processor has to periodically poll or check the status/readiness of the device and can perform data transfer only when the device 'is ready. In polling technique the processor time is wasted, because the processor has to suspend its work and check the status of the device in predefined intervals.

Alternatively, if the device interrupts the processor to initiate a data transfer whenever it is ready then the processor time is effectively utilized because the processor need not suspend its work and check the status of the device in predefined intervals.

For an example, consider the data transfer from a keyboard to the processor. Normally a keyboard has to be checked by the processor once in every 10 milli seconds for a key press. Therefore once in every 10 milli seconds the processor has to suspend its work and then check the keyboard for a valid key code. Alternatively, the keyboard can interrupt the processor, whenever a key is pressed and a valid key code is generated. In this way the processor need not waste its time to check the keyboard once in every 10 milli seconds.

4.4.3 Classification of Interrupts

In general the interrupts can be classified in the following three ways :

1. Hardware and software interrupts
2. Vectored and Non Vectored interrupt:
3. Maskable and Non Maskable interrupts.

The interrupts initiated by external hardware by sending an appropriate signal to the interrupt pin of the processor is called hardware interrupt. The 8086 processor has two interrupt pins INTR and NMI. The interrupts initiated by applying appropriate signal to these pins are called hardware interrupts of 8086.

The software interrupts are program instructions. These instructions are inserted at desired locations in a program. While running a program, if software interrupt instruction is encountered then the processor initiates an interrupt. The 8086 processor has 256 types of software interrupts. The software interrupt instruction is INT n, where n is the type number in the range 0 to 255.

When an interrupt signal is accepted by the processor, if the program control automatically branches to a specific address (called vector address) then the interrupt is called vectored interrupt. The automatic branching to vector address is predefined by the manufacturer of processors. (In these vector addresses the interrupt service subroutines (ISR) are stored). In non-vectored interrupts the interrupting device should supply the address of the ISR to be executed in response to the interrupt. All the 8086 interrupts are

vectored interrupts. The vector address for an 8086 interrupt is obtained from a vector table implemented in the first 1kb memory space (00000h to 03FFFh).

The processor has the facility for accepting or rejecting hardware interrupts. Programming the processor to reject an interrupt is referred to as masking or disabling and programming the processor to accept an interrupt is referred to as unmasking or enabling. In 8086 the interrupt flag (IF) can be set to one to unmask or enable all hardware interrupts and IF is cleared to zero to mask or disable a hardware interrupts except NMI.

The interrupts whose request can be either accepted or rejected by the processor are called maskable interrupts. The interrupts whose request has to be definitely accepted (or cannot be rejected) by the processor are called non-maskable interrupts. Whenever a request is made by non-maskable interrupt, the processor has to definitely accept that request and service that interrupt by suspending its current program and executing an ISR. In 8086 processor all the hardware interrupts initiated through INTR pin are maskable by clearing interrupt flag (IF). The interrupt initiated through NMI pin and all software interrupts are non-maskable.

4.4.4 Sources of Interrupts in 8086

An interrupt in 8086 can come from one of the following three sources.

1. One source is from an external signal applied to NMI or INTR input pin of the processor. The interrupts initiated by applying appropriate signals to these input pins are called hardware interrupts.
2. A second source of an interrupt is execution of the interrupt instruction "INT n", where n is the type number. The interrupts initiated by "INT n" instructions are called software interrupts.
3. The third source of an interrupt is from some condition produced in the 8086 by the execution of an instruction. An example of this type of interrupt is divide by zero interrupt. Program execution will be automatically interrupted if you attempt to divide an operand by zero. Such conditional interrupts are also known as exceptions.

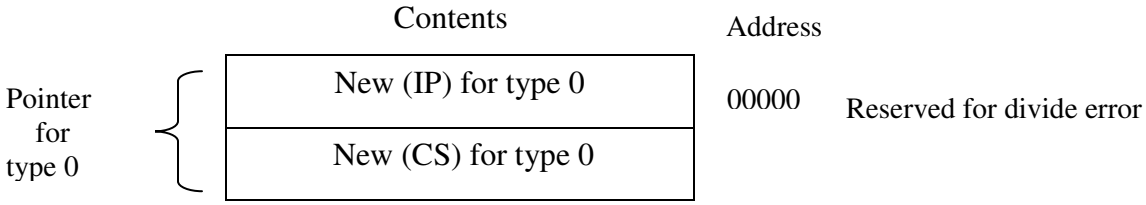
4.4.5 Interrupts of 8086

The 8086 microprocessor has 256 types of interrupts. INTEL has assigned a type number to each interrupt. The type numbers are in the range of 0 to 255. The 8086 processor has dual facility of initiating these 256 interrupts. The interrupts can be initiated either by executing "INT n" instruction where n is the type number or the interrupt can be initiated by sending an appropriate signal to INTR input pin of the processor.

For the interrupts initiated by software instruction" INT n ", the type number is specified by the instruction itself. When the interrupt is initiated through INTR pin, then the processor runs an interrupt acknowledge cycle to get the type number. (i.e., the interrupting device should supply the type number through D_0- D_7 lines when the processor requests for the same through interrupt acknowledge cycle).

The kinds of interrupts and their designated types are summarized in fig. 4.5 by illustrating the layout of their pointers within the memory. Only the first five types have explicit definitions; the other types may be used by interrupt instructions or external interrupts. From the figure it is seen that the type associated with a division error interrupt is 0. Therefore, if a division by 0 is attempted, the processor will push the current contents of the PSW, CS and IP into the stack, fill the IP and CS registers from the addresses 00000 to 00003, and continue executing at the address indicated by the new contents of IP and CS. A division error interrupt occurs any time a DIV or IDIV instruction is executed with the quotient exceeding the range, regardless of the IF (Interrupt flag) and TF (Trap flag) status.

The type 1 interrupt is the single-step interrupt (Trap interrupt) and is the only interrupt controlled by the TF flag. If the TF flag is enabled, then an interrupt will occur at the end of the next instruction that will cause a branch to the location indicated by the contents of 00004H to 00007H.The single step interrupt is used primarily for debugging which gives the programmer a snapshot of his program after each instruction is executed.



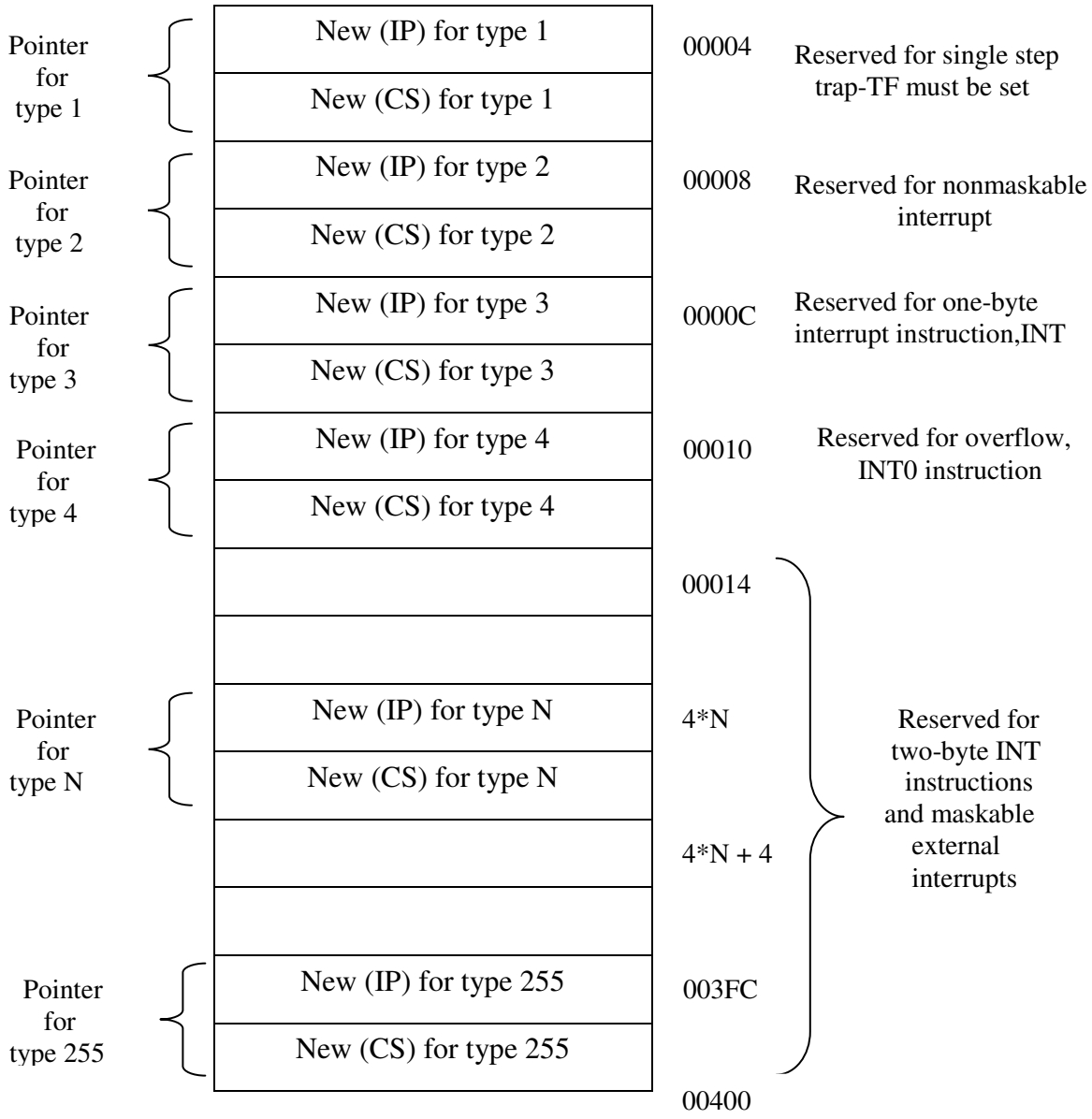


Fig. 4.5 Organisation of Interrupt vector table in 8086

The type 2 interrupt is the nonmaskable external interrupt. It is the only external interrupt that can occur regardless of the IF flag setting. It is caused by a signal sent to the CPU through the nonmaskable interrupt pin.

The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts. The interrupt instructions are summarized below and their interrupts are not controlled by the IF flag.

Name	Mnemonics	Description
Interrupt with Type	INT TYPE	(SP) ← (SP) – 2 ((SP)+1:(SP)) ← (PSW) (SP) ← (SP) – 2 ((SP)+1:(SP)) ← (CS) ((SP) ← (SP) – 2 ((SP)+1:(SP)) ← (IP) (IP) ← (TYPE * 4) (CS) ← (TYPE * 4) + 2
One byte interrupt	INT	(SP) ← (SP) – 2 ((SP)+1:(SP)) ← (PSW) (SP) ← (SP) – 2 ((SP)+1:(SP)) ← (CS) ((SP) ← (SP) – 2 ((SP)+1:(SP)) ← (IP) (IP) ← (0CH) (CS) ← (0EH)
Interrupt on Overflow	INTO	If (OF) = 1, then (SP) ← (SP) – 2 ((SP)+1:(SP)) ← (PSW) (SP) ← (SP) – 2 ((SP)+1:(SP)) ← (CS) ((SP) ← (SP) – 2 ((SP)+1:(SP)) ← (IP) (IP) ← (10H) (CS) ← (12H)

Return from Interrupt	IRET	(IP) ⟵ ((SP)+1:(SP)) (SP) ⟵ (SP) + 2 (CS) ⟵ ((SP)+1:(SP)) (SP) ⟵ (SP) + 2 (PSW) ⟵ ((SP)+1:(SP)) (SP) ⟵ (SP) + 2

IRET is used to return from an interrupt service routine. It is similar to the RET instruction except that it pops the original contents of the PSW from the stack as well as the return address.

The INT instruction has one of the forms

INT

or INT Type

The INT instruction is also often used as a debugging aid in cases where single stepping provides more detail than is wanted. By inserting INT instructions at key points, called *breakpoints.* Within a program a programmer can use an interrupt routine to provide messages and other information at these points. Hence the 1 byte INT instruction (Type 3 interrupt) is also referred to as *breakpoint interrupt*.

The INTO instruction has type 4 and causes an interrupt if and only if the OF flag is set to 1. It is often placed just after an arithmetic instruction so that special processing will be done if the instruction causes an overflow. Unlike a divide-by-zero fault, an overflow condition does not cause an interrupt automatically; the interrupt must be explicitly specified by the INTO instruction. The remaining interrupt types correspond to interrupts instructions imbedded in the interrupt program or to external interrupts.

4.4.6 Interrupt Programming

Program 4.3

Write a program to a create a file RESULT and store in it 500H bytes from the memory block starting at 1000:1000, if either an interrupt occurs at INTR pin with Type 0AH or an instruction equivalent to the above interrupt is executed.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
FILENAME DB "RESULT", "$"
MESSAGE DB "FILE WASN'T CREATED SUCCESSFULLY" ,
                0AH,0DH,"$"
DATA ENDS
CODE SEGMENT
START  : MOV AX,CODE
          MOV DS,AX          ; Set DS at code for setting IVT
          MOV DX,OFFSET ISR0A ; Set DX at offset of ISR0A.
          MOV AX,250AH    ;Set IVT using function value 250AH
         INT 21H             ; in AX under INT 21H
         MOV DX,OFFSET FILENAME ; Set pointer to filename.
         MOV AX,DATA      ; Set the DS at DATA for filename.
         MOV DS,AX
         MOV CX,00H
        MOV AH,3CH         ; Create file with the filename 'RESULT'
       INT 21H
       JNC FURTHER         ; If no carry, create operation is successful
       MOV DX,OFFSET MESSAGE ; else display message
       MOV AH,09H
       INT 21H
       JMP STOP
       FURTHER: INT 0AH   ; If the file is created successfully,
                   MOV AH,4CH ; write into it and return to DOS prompt
```

```
                INT 21H
        ISR0A PROC NEAR
        MOV BX,AX ; Take file handle in BX,
        MOV CX,500H ; Byte count in CX
        MOV DX,1000H ; Offset of block in DX
        MOV AX,1000H  ; Segment value of block
        MOV DS,AX       ; in DS
        MOV AH,40H      ; Write in the file and
        INT 21H         ; return
        ISR0AH ENDP
        CODE ENDS
        END START
```

Program 4.4

Write a program that gives display 'IRT2 is OK' if a hardware Signal appears on IRQ_2 pin and 'IRT3 is OK' if it appears on IRQ_3 pin of PC IO channel.

```
ASSUME CS:CODE, DS:DATA
DATA SEGMENT
MSG1 DB "IRT2 is OK",0AH,0DH,"$"
MSG2 DB "IRT3 is OK",0AH,0DH,"$"
DATA ENDS
CODE SEGMENT
   START : MOV AX,CODE
           MOV DS,AX                    ; Set IVT for Type 0AH
           MOV DX, OFFSET ISR1
           MOV AX,250AH      ; IRQ2 is equivalent to Type 0AH
           INT 21H
           MOV DX, OFFSET ISR2 ; Set IVT for Type 0BH
          MOV AX,250BH       ; IRQ3 is equivalent to Type 0BH
          INT 21H
```

```
HERE: JUMP HERE
      ISR1 and ISR2 display the message
      ISR1 PROC LOCAL
      MOV AX,DATA
      MOV DS,AX
      MOV DX,OFFSET MSG1 ; Display message MSG1
      MOV AH,09H
      INT 21H
      IRET
      ISR1 ENDP
      ISR1 and ISR2 display the message
      ISR1 PROC LOCAL
      MOV AX,DATA
      MOV DS,AX
      MOV DX,OFFSET MSG1 ; Display message MSG1
      MOV AH,09H
      INT 21H
      IRET
      ISR1 ENDP
      ISR2 PROC LOCAL
     MOV AX,DATA
      MOV DS,AX
      MOV AX,OFFSET MSG2     ; Display message MSG2
      MOV AH,09H
      INT 21H
      IRET
  ISR2 ENDS
  END START
ISR2 PROC LOCAL
      MOV AX,DATA
      MOV DS,AX
```

```
        MOV AX,OFFSET MSG2     ; Display message MSG2
        MOV AH,09H
        INT 21H
        IRET
    ISR2 ENDS
     END START
```

4.5 MACROS

4.5.1 Disadvantages of Procedure

1. Linkage associated with them.
2. It sometimes requires more code to program the linkage than is needed to perform the task. If this is the case, a procedure may not save memory and execution time is considerably increased.
3. Hence a means is needed for providing the programming ease of a procedure while avoiding the linkage. This need is fulfilled by **Macros.**

Macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each reference.

A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced.Therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved. The code that is to be repeated is called the prototype code. The prototype code along with the statements for referencing and terminating is called the macro definition.

Once a macro is defined, it can be inserted at various points in the program by using macro calls. When a macro call is encountered by the assembler, the assembler replaces the call with the macro code. Insertion of the macro code by the assembler for a macro call is referred to as a macro expansion.

In order to allow the prototype code to be used in a variety of situations, macro definition and the prototype code can use dummy parameters which can be replaced by the actual parameters when the macro is expanded. During a macro expansion, the first

actual parameter replaces the first dummy parameter in the prototype code, the second actual parameter replaces the second dummy parameter, and so on.

4.5.1 ASM-86 Macro Facilities

The macro definition is constructed as follows :

```
%*DEFINE(Macro name(Dummy parameter list))
  (
        Prototype code
  )
```

Macro name has to begin with a letter and can contain letters, numbers and underscore characters. Dummy parameters in the parameter list should be separated by commas. Each dummy parameter appearing in the prototype code should be preceded by a % character. Consider an example that shows the definition of macro for multiplying 2 word operands and storing the result which does not exceed 16 bit.

A macro call has the form

```
%Macro name (Actual parameter list)
```

with the actual parameters being separated by commas.

```
%MULTIPLY (CX,VAR,XYZ[BX]
```

Above macro call results in following set of codes.

```
PUSH DX
PUSH AX
MOV AX,CX
IMUL VAR
MOV XYZ[BX],AX
POP AX
POP DX
```

It is possible to define a macro with no dummy parameters, but in this case the call must not include any parameters. Consider a macro for pushing the contents at beginning of a procedure.

Macro definition consists of

```
%*DEFINE(SAVEREG)
    ( PUSH AX
      PUSH BX
      PUSH CX
     PUSH DX
    )
```

This macro is called using the statement

```
%SAVEREG
```

The above macro can be called at the beginning of the each procedure to save the register contents. A similar macro could be used to restore the register contents at the end of each procedure.

```
%*DEFINE (RESTORE)
    ( POP DX
      POP CX
      POP BX
      POP AX
       )
```

4.5.2 Local Labels

Consider a macro called ABSOL which makes use of labels. This macro is used to replace the operand by its absolute value.

```
%*DEFINE(ABSOL(OPER))
  ( CMP %OPER,0
    JGE NEXT
    NEG %OPER
NEXT: NOP
  )
```

When the macro ABSOL is called for the first time, the label NEXT will appear in the program and, therefore it becomes defined. Any subsequent call will cause NEXT to be redefined. This will result in an error during assembly process because NEXT ha been

associated with more than one location. One solution to this problem would be to have NEXT replaced by a dummy parameter for the label. This would require the programmer to keep track of dummy parameters used. One solution to this problem is the use of **Local Labels.**

Local labels are special labels that will have suffixes that gets incremented each time the macros are called. These suffixes are two digit numbers that gets incremented by one starting from zero. Labels can be declared as local label by attaching a prefix **Local.**

Local List of Local labels at the end of first statement in the macro definition. Consider a macro which makes use local labels.

```
%*DEFINE(ABSOL(OPER)) LOCAL NEXT
        ( CMP %OPER,0
          JGE %NEXT
           NEG %OPER
      %NEXT:NOP
        )
```

If this macro is called twice using **%ABSOL(VAR)** and **%ABSOL(BX)** would result in following set of codes.

```
            CMP VAR,0
            JGE NEXT00
            NEG VAR
NEXT00: NOP
              .
            CMP BX,0
            JGE NEXT01
            NEG VAR
NEXT01: NOP
```

4.5.3 Nested Macros

It is possible for a macro call to appear within a macro definition. This is referred to as **Macro nesting.** The limitation of nested macros is that all macros included in the definition of a given macro must be defined before the given macro is called.

```
%*DEFINE(DIFSOR(OPR1,OPR2,RESULT))
        ( PUSH DX
          PUSH AX
         %DIF(%OPR1,%OPR2)
          IMUL AX
          MOV %RESULT,AX
          POP AX
          POP DX
    )
%*DEFINE(DIF(X,Y))
        MOV AX,%X
        SUB AX,%Y
            .
            .
    %DIFSOR(VAR1,VAR2,ERROR)
            .
            .
```

This results in following set of codes

```
        PUSH DX
        PUSH AX
        MOV AX,VAR1
        SUB AX,VAR2
        IMUL AX
        MOV ERROR,AX
        POP AX
        POP DX
```

Program 4.5

A program using Macro for saving the contents of GPRs in the stack.

```
        ASSUME CS:CODE, DS:DATA
        DATA SEGMENT
```

```
        SAVEREG MACRO
        PUSH AX
        PUSH BX
        PUSH CX
        PUSH DX
    ENDM
     DATA ENDS
CODE SEGMENT
    START : MOV AX,DATA
            MOV DS,AX
            MOV AX,1234H
            MOV BX,2345H
            MOV CX,3456H
            MOV DX,4567H
            SAVEREG
            MOV AH,4CH
             INT 21H
            CODE ENDS
            END START
```

- This program executes following set of codes

```
            MOV AX,DATA
            MOV DS,AX
            MOV AX,1234H
            MOV BX,2345H
            MOV CX,3456H
            MOV DX,4567H
            PUSH AX
            PUSH BX
            PUSH CX
            PUSH DX
            MOV AH,4CH
             INT 21H
```