1.5 MODULAR PROGRAMMING

Complex programs are divided into many parts and each sub-part are known as modules. All the modules perform a well-defined task. Formulation of computer code using a module is known as modular programming.

The reasons for breaking a program into small parts are

- Modules are easier to understand.
- Different modules can be assigned to different programmers.
- The debugging and testing can be done in a more orderly fashion.
- Documentation can be more easily understood.
- Modifications may be localized.

Most assembler languages are used in modularization process in three ways such as,

1. Allow data to be structured so that they can be accessed by several modules.

2. Provide for procedures or subroutines.

3. Permit sections of code known as macros.

To perform modular programming, the following tasks must be performed.

- Linking and relocation
- Stack Operation
- Procedures
- Interrupt process

LINKING AND RELOCATION

The assembly language program can be written with an ordinary text editors such as word star, editor etc. The assembly language program text is an input to the assembler. The assembler translates assembly language statements to their binary equivalent known as object code. During assembling process assembler checks for syntax errors and displays them before giving object code module. The object code module contains the information about where the program or module to beloaded in memory. If the object module is to be linked with other separate modules then it contains additional linkage information.

At link time, separately assembled modules are combined into one single load module by the linker. The linker also SUBs any required initialization or finalization code

to allow the OS to start the program running and to return control to OS after the program is completed. At load time, the program loader copies the program into computer main memory and at execution time, the program execution begins. If the modules in the program they are assembled separately, then there is one main module and other modules. This main module has the first instruction to be executed and it is terminated by an END statement with entry point satisfied. Other modules are terminated by an END statement with no operand.

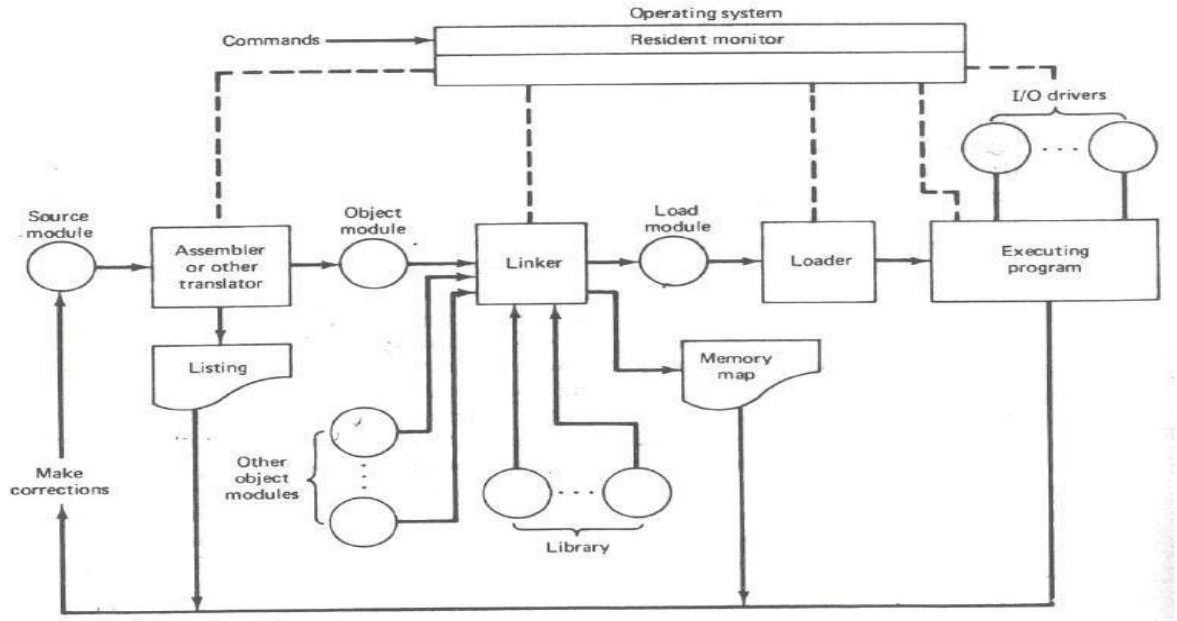

Figure 1.5.1 Creation and Execution of Assembly language program

[Source: "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design" by Yu-Cheng Liu, Glenn A.Gibson]

SEGMENT COMBINATION

In addition to the linker commands, the assembler provides a means of regulating the waysegments in different object modules are organized by the linker. Segments with same name are joined together by using the modifiers attached to theSEGMENT 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 individuallyspecified 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.

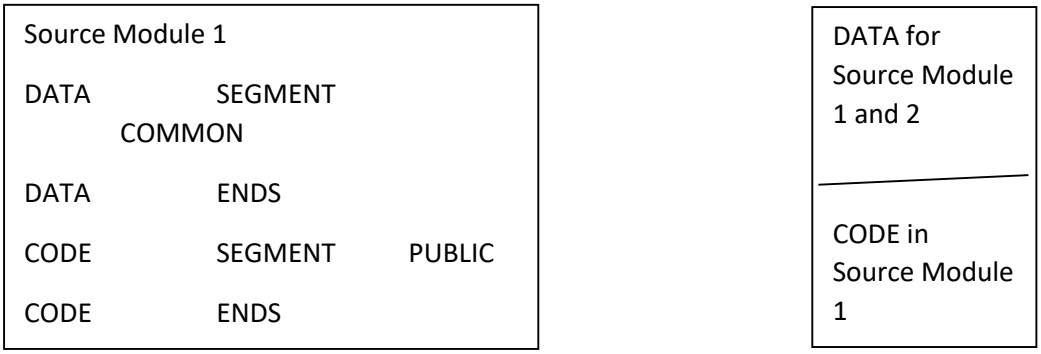

Figure 1.5.2 Segment combinations resulting from the PUBLIC and Common Combination types

[Source: "Microcomputer Systems: The 8086 / 8088 Family - Architecture, Programming and Design" by Yu-Cheng Liu, Glenn A.Gibson]

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* relativeto the module. Two lists are implemented by the EXTRN and PUBLIC directives, which have the forms:

```
EXTRN Identifier Type….. Identifier Type
And
```

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 willbe 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.

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. 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 Addressing the CALL instruction i.e. starting Address of the Procedure. Then the procedure is executed.

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.

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, andthen 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

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:

- The same code segment as the statement that calls it.
- A code segment that is different from the one containing the statement that calls it, but in thesame source module as the calling statement.
- 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.

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 callingprogram.

```
MSK PROC NEARPUSH AX PUSH BX PUSHCX
  POP CXPOP BXPOP AXRET
MSK ENDP
```

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 tothe variables directly. When the procedure is in a separate source module it can still refer to the source moduledirectly provided that the calling program contains the directive PUBLIC ARY, COUNT, SUM EXTRN ARY: WORD, COUNT: WORD, SUM: WORD

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

Eg. Recursive procedure to compute the factorial

Disadvantages of Procedure

a. Linkage associated with them.
b. It sometimes requires more code to program the linkage than is needed to perform the task. Ifthis is the case, a procedure may not save memory and execution time is considerably increased.

MACROS

Macros are needed for providing the programming ease of a procedure while avoiding the linkage. 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 macrocalls. 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. 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 soon.

A macro call has the form

%Macro name (Actual parameter list) with the actual parameters being separated bycommas.

%MULTIPLY (CX, VAR, XYZ[BX]

Above macro call results in following set of codes.

PUSH DX

PUSH AX MOV AX,CXIMUL VAR

MOV XYZ[BX],AXPOP AX

POPDX

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.