Procedures

Definition:

A procedure is a reusable section of the software that is stored in memory once, used as often as necessary.

Calling a procedure :

The CALL instruction links to the procedure and the RET 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.

Procedures in MASM:

A procedure begins with the **PROC** directive and ends with the **ENDP** directive. Each directive appears with the name of the procedure. The directive PROC is followed by the procedure type: **NEAR** (intra-segment) or **FAR** (inter-segment).

Note: In MASM version 6.X NEAR or FAR procedures may be followed by the **USES** directive. The **USES** directive allows any number of registers to be automatically pushed onto the stack and popped from the stack within the procedure.

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 the return address in 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.

Instruction	Example	Effect	
CALL	CALL SQRT	[SP-1] ✍ 34H [SP-2] ✍ 5BH	SP ✍ SP-2 IP ✍ 34A0H
RET	RET	LSB(IP) ✍ [SP] MSB(IP) ✍ [SP+1]	SP ✍ SP + 2

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

Table 14. 1: Summary of the Subroutine Handling Instructions

Note:
Procedures that are to be used by all program (*global*) should be written as FAR procedures. Procedures that are used within the same program are normally defined as NEAR procedures.

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.

Far CALL:

The FAR CALL calls 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 contents of both IP and CS are put on the stack before jumping to the address indicated by bytes 2 to 5 of the instruction. A variant of the FAR CALL is CALLF but should be avoided.

Calls With Register Operand:

CALLs may 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 indicated by 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.

Calls 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. 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.

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 registers which need to be saved and are used by the subroutine, should be saved and retrieved afterwards.