Macros and Procedures

Macros

Overview
A macro is a named block of assembly language statements. Once defined, it can be invoked (called) as many times in a program as you wish. When you invoke a macro, a copy of its code is inserted directly into the program at the location where it was invoked. This type of automatic code insertion is also known as inline expansion. It is customary to refer to calling a macro, although technically there is no CALL instruction involved.

Declaring Macros are defined directly at the beginning of a source program, or they are placed in a separate file and copied into a program by an INCLUDE directive. Macros are expanded during the assembler’s preprocessing step. In this step, the preprocessor reads a macro definition and scans the remaining source code in the program. At every point where the macro is called, the assembler inserts a copy of the macro’s source code into the program. A macro definition must be found by the assembler before trying to assemble any calls of the macro. If a program defines a macro but never calls it, the macro code does not appear in the compiled program.

Defining Macros
A macro is defined using the MACRO and ENDM directives. The syntax is:

```
macroname MACRO parameter-1, parameter-2...statement-list
.
.
ENDM
```

There is no fixed rule regarding indentation, but we recommend that you indent statements between `macroname` and ENDM. You might also want to prefix macro names with the letter m, creating recognizable names such as `mPutChar`, `mWriteString`, and `mGotoxy`. The statements between the MACRO and ENDM directives are not assembled until the macro is called. There can be any number of parameters in the macro definition, separated by commas.

Parameters: Macro parameters are named placeholders for text arguments passed to the caller. The arguments may in fact be integers, variable names, or other values, but the preprocessor treats them as text. Parameters are not typed, so the preprocessor does not check argument types to see whether they are correct. If a type mismatch occurs, it is caught by the assembler after the macro has been expanded.

Example: The following `Putchar` macro receives a single input parameter called `char` and displays it on the console:

```
Putchar MACRO char
mov dl,char
mov ah,02h
int 21h
ENDM
```
Invoking Macros
A macro is called (invoked) by inserting its name in the program, possibly followed by macro arguments. The syntax for calling a macro is:

```
macroname argument-1, argument-2, ...
```

`Macroname` must be the name of a macro defined prior to this point in the source code. Each argument is a text value that replaces a parameter in the macro. The order of arguments must correspond to the order of parameters, but the number of arguments does not have to match the number of parameters. If too many arguments are passed, the assembler issues a warning. If too few arguments are passed to a macro, the unfilled parameters are left blank.

Procedures
Defining a Procedure
Informally, we can define a procedure as a named block of statements that ends in a return statement.

A procedure is declared using the PROC and ENDP directives. It must be assigned a name (a valid identifier). A procedure must be written in the end of the code segment (before `end` directive), and it can't receive any parameter.

When you create a procedure other than your program’s startup procedure, end it with a RET instruction. RET forces the CPU to return to the location from where the procedure was called. A procedure is defined using the PROC and ENDP directives.

The syntax is:

```
procedurename PROC
  ...
ret
procedurename ENDP
```

Example: Sum of Three Integers
Let’s create a procedure named `SumOf` that calculates the sum of three 16-bit integers. We will assume that relevant integers are assigned to AX, BX, and CX before the procedure is called. The procedure returns the sum in AX:

```
SumOf PROC
  add ax,bx
  add ax,cx
  ret
SumOf ENDP
```

Invoking procedures
A procedure is called (invoked) by inserting its name in the program Preceded by (call). The syntax for calling a procedure is:

```
Call procedurename
```

CALL and RET Instructions
The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location. The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called. Mechanically speaking, the CALL instruction pushes its return address on the stack and copies the called procedure’s address into the instruction pointer. When the procedure is ready to return, its RET instruction pops the return address from the stack into the instruction pointer. In 32-bit mode, the CPU executes
the instruction in memory pointed to by EIP (instruction pointer register). In 16-bit mode, IP points to the instruction.