

MACRO AND PROGRAM TESTING

Every macro has a single **macro declaration**, and zero or more **macro invocations**. The invocations must occur physically after the declaration. (The file PCMAC.INC is essentially just a list of macro declarations.)

Macro Declarations

```

MacroName      MACRO      Macro Parameter List
                ....
                Prototype of the macro
                ....
                ENDM

```

The **prototype** of the macro is a sequence of 'real' machine instructions (or other macro invocations) which will replace every invocation of *MacroName*.

A simple form of `_PutStr` macro could be declared as follows:

```

_PutStr      MACRO      aString
              mov  dx, OFFSET aString
              mov  ah, 09h
              int  21h
              ENDM

```

A macro invocation has the form:

```

MacroName      Actual Parameters to the macro

```

Each time the assembler finds a macro invocation; it replaces the formal arguments with the actual parameters and then deletes the macro invocation and replaces it with the new version and replaces it with the new version of macro body.

Example Invocation

```
_PutStr myMessage
```

Example after Macro Expansion

```

mov dx, OFFSET myMessage
mov ah, 09h
int 21h

```

This example does not save us a lot of work, but we don't have to worry about remembering what it takes (registers, codes, etc) to make it work. It is easier to remember `_PutStr` and the label we wish to display.

Macro Expansion

This macro is a "safe" macro, because it saves and restores affected registers. This is good sometimes, but it will always add the overhead of a push/pop pair.

```
AddUp MACRO    A, B, Sum
    push        ax
    mov         ax, A
    add         ax, B
    mov         SUM, ax
    pop         ax
ENDM
```

Now to invoke it:

```
AddUp 3, bx, Total
```

This will become:

```
push ax
mov ax, 3
add ax, bx
mov Total, ax
pop ax
```

Notice that in this invocation, we used a constant, a memory location, and a register. The sum gets stored in the memory location with the label Total. Additionally, the actual parameters are separated by commas, blanks, and tabs.

If a macro invocation has more arguments than the declaration, the extra arguments are ignored. If it has fewer arguments, the extra parameters in the declaration are set to the empty string. That is what happens in our version of `_PutStr`, there is one more parameter than we have used.

Parameters Parentheses

Since blanks, tabs, and commas act as parameter separators, they must be handled differently when we want to embed them in an argument. Suppose we have the macro:

```
DecBytes MACRO A, B, Symbol
    Symbol DB    A DUP (B)
ENDM
```

If we invoke it with:

```
DecBytes 5 DUP (3)
```

We have actually have arguments of "5", "DUP", and "(3)". We end up with:

```
(3) DB 5 DUP (DUP)
```

However, if we invoke it with:

```
DecBytes 100, <5 DUP (3)>
```

We end up with:

```
DB 100 DUP (5 DUP 3)
```

Make sure you remember that blanks are separators!

```
10 dup(5)  is two parameters
10 dup (5)  is three parameters
10 dup ( 5 ) is five parameters
```

Surround complicated macro parameters with <> parentheses

Whenever a macro parameter is passed as a parameter to an inner macro, it should be enclosed in <> parentheses.

Macros are more dangerous than subprograms, because some register usage is not obvious.

When in doubt, save and restore the registers.

A safe version is:

```
bMac MACRO aPar
    aMac    <aPar>
    ENDM
```

Local Symbols

When a label is defined inside a macro, each time the macro expanded, the same label is reused. This is an error condition. The way to avoid that is to use local symbols.

```
Max    MACRO A, B
        LOCAL noChange
        mov    ax, A
        cmp    ax, B
        jge    noChange
        mov    ax, B
noChange:
        ENDM
```

Assembly Listings

To see the actual code produced by the macros, you can look at the assembly listing with the command:

```
masm afile,,afile;
or
ml /FI afile.asm
```

Pseudo-Macros for Repetition

There are three pseudo-macros which expand a prototype repeatedly:

Pseudo-Macro	Meaning
REPT n	REPeaT n times
IRP	Indefinite RePeat
IRPC	Indefinite RePeat Characters

The simplest of these is the pseudo-macro **REPT** which merely repeats the prototype **n** times. Without any parameters, we could have a macro:

```
REPT 20
DB 5, 3, ?, 18, 18
ENDM
```

This is the equivalent of:

```
DB 20 DUP ( 5, 3, ?, 18, 18 )
```

We can write a macro for a number of times, but let the computer do the counting:

```
IPR parm, <x1, x2, ... xn >
...prototype containing &param&....
ENDM
```

An example is:

```
IRP aReg, <ax, bx, cx, dx>
push &aReg&
ENDM
```

This expands to:

```
push ax
push bx
push cx
push dx
```

Actually, we could do that one in a different manner, since there is really only one character difference:

```
IRPC regLet, abcd
push &regLet&x
ENDM
```