MACROS for I/O

As you have probably noticed, writing DOS calls can become tedious. Much of the code is repetitive, and each call has its own function code and register usage. You are probably used to dealing with complexity by making a repeated code a **procedure** or **function.**

Assembly language allows you to use a single invented name to represent frequently used sequence of instructions. This name is used like an ordinary machine instruction, and is called a **macro-instruction**, or **macro** for short. **Macro** means 'large' and is intended to suggest grouping several instructions into a single large instruction. The group can contain **ordinary machine operations**, **pseudo-operations**, or even other **macro instructions.**

When an assembler encounters the name of a **macro**, it replaces it with the sequence of instructions the macro consists of. The replacement process is called **expanding the macro**. This is analogous to the **#define** statement in C. We will make use of an inbuilt package of macros that is supplied by **MASM** called **PCMAC.INC**. Any source program that uses **PCMAC.INC** must contain the following line before using any of its macros:

```
INCLUDE PCMAC.INC
```

The statement above assumes that the **PCMAC.INC** file is in the current directory. The full path of the file can also be given:

```
INCLUDE C:\MASM615\INCLUDE\PCMAC.INC
```

The simplest macro we will use is **_Begin**, which acts as a substitute for the **standard code** in the **FIRST.ASM** program replacing:

```
mov     ax, @data
mov     ds, ax
```

Two more complicated macros are **_PutStr**, which is the substitute for the **DOS 9h** call and **_Exit** which is the substitute for the **DOS 4ch** call. They are in the form:

```
_PutStr label-of-'$' -terminated-string
```

and

```
_Exit           program-return-value
```

program-return-value is optional and if omitted, 0 will be used. The first letter is usually capitalized to distinguish them from machine instructions and pseudo-operations and all macros in **PCMAC.INC** starts with an underscore to distinguish them from user-created macros.

Hence **FIRST.ASM** can be rewritten as:

```
;; FIRST.ASM – Our first Assembly Language Program. This program
;; displays the line " Hello, my name is nefertum" on the screen

INCLUDE C:\MASM615\INCLUDE\PCMAC.INC

        .MODELSMALL
        .586            ; allows Pentium instructions. Must come after .MODEL
        .STACK  100h
```

```
                .DATA
Message         DB      'Hello, my name is Rishi Heerasing', 13, 10, '$'

                .CODE
Hello           PROC
                _Begin
                _PutStr Message
                _Exit   0       ;Return to DOS with (normal) return code 0
Hello           ENDP
                END     Hello           ; tells where to start execution
```

❖ THE DOS DISPLAY CHARACTER CALL

You may be wondering how to display the "$" character, since **_PutStr** stops when it sees a '$' and doesn't display it. In fact there is NO WAY to display a "$" with **_PutStr**. (Recall Ex6Msg and Ex7Msg which print exactly the same thing, and neither prints the whole message. The easiest way to display a "$" is to use another DOS call, number **2h**, which displays the single character it finds in register **dl**. This can be achieved by the following:

```
mov             dl, '$'         ;copy "$" in dl
mov             ah, 2h          ;call the DOS function
int             21h             ;Return to OS
```

The above code has a corresponding macro, **_PutCh**, the syntax is

```
_PutCh          '$'
```

The **_PutCh** macro can be used to display one or more characters by listing them in the operand position. For instance, we could display a carriage return/line feed pair by coding,

```
_PutCh          13, 10
```

which generates two DOS 2h calls. (Note: **_PutStr Message1, Message2** is not allowed)

❖ MAGIC NUMBERS

It is usually poor programming practice to use **magic numbers** in a program. **Magic numbers** are **constants** (other than 0 and 1) that have some significance in the program. Some examples of magic numbers in our earlier programs were **10, 13**, **9h** and **4ch**. It is better to give such numbers symbolic names. **C** does it using the **#define** statement but our assembler does it with **EQU** (short for EQUate) pseudo-op. We could therefore code the following:

```
CR              EQU     13              ;equate CR to Carriage Return
LF              EQU     10              ;equate LF to Line Feed
Message         DB      'Hello, my name is nefertum', CR, LF, '$'
```

We can even set an **equate** for "$":

```
                MsgEnd          EQU             '$'
The code
                A               EQU             B
```

is roughly equivalent to the C code

```
#define   A     B       e.g. #define  PI  3.142
```

B can be replaced by an expression. Thus

```
A           EQU         10
B           EQU         A + 1    ; is LEGAL: sets B to 11
```

Forward references are allowed:

```
A           EQU         B + 1    ; is LEGAL: A eventually becomes 11
B           EQU         10
```

But uses of A **cannot occur before** B is defined and **circular equates** are forbidden.
Once a symbol has been defined with the EQU pseudo-op, it cannot be redefined.

There are several advantages to using EQUates:

- They make the program easier to read by documenting the meaning of constants.
- They make the program easier to change as changes needs to be effected in one place only.
- Different versions of a program can be specified by the values of one or more EQUates, set at the beginning of the program.
- The points above are even more important in programs where a single constant has two different meanings.

As an example of the last point consider that the program also used the number 13 as an unlucky number and declared as follows:

```
Unlucky     EQU         13
CR          EQU         13
```

Then if at some later stage, we decided to change the unlucky 13's (but not **Carriage Return** 13's); it would be easy to do so.

❖ NUMERIC I/O

As you have probably realised, programming I/O of numbers in Assembly turns out to be more difficult and cryptic than expected. That is why we use high level languages whenever we can. Since reading and displaying numbers is done repeatedly, a **library** called **UTIL.LIB** will be used. This library contains sub-procedures to do numeric I/O and other useful things.

We will look at six simple numeric I/O procedures present in that **library**:

GetDec	reads a **decimal** integer (with optional "-"sign) from the keyboard and returns with its value (as a **binary**) in **AX**. **GetDec** reads characters until it encounters one that cannot belong to the number.
GetDDec	is similar to **GetDec** but reads in a **double word** (32-bit) in **EAX**.
PutDec	displays the (**binary**) number in **AX** as a **decimal** number (with – sign if negative) at the cursor in the Prompt Window.
PutDDec	is similar to **PutDec** but displays the number in EAX instead.
PutHex	displays the (**binary**) number in **AX** as a **hexadecimal** number at the cursor in the Prompt Window (in exactly four characters, with no trailing 'h' or extra leading '0').
PutDHex	is similar to **PutHex** but displays the number in **EAX** as exactly eight characters.

A program that uses these procedures must include the **EXTRN** pseudo-operation line immediately after the **.CODE** pseudo-op:

```
                    Accessing Library Procedures

                .CODE
                EXTRN   GetDec : NEAR, PutDec : NEAR, etc...
ProcName        PROC    ...
```

EXTRN stands for **EXTeRNal**, indicating that these procedures are defined elsewhere. The **EXTRN** procedure only needs to list the procedures you are actually using. **NEAR** will be discussed later.

The difference between **EXTRN** and **INCLUDE** is as follows: **INCLUDE** causes the assembler to insert a file of source text into the program while assembling it. **EXTRN** tells the assembler that the linker will find an already assembled **PROC** of this name in another of the object files it is told to link.
All of the following procedures are executed by using the call instruction. To display a number at the cursor position in the Prompt Window:

Displaying Numbers

```
;Display a number in decimal
        mov     ax, number-to-be-displayed
        call    putDec          ;display number in decimal
;Display a LARGE number in decimal
        mov     eax, number-to-be-displayed
        call    putDDec                 ;display number in decimal
;Display a number in hexadecimal
        mov     ax, number-to-be-displayed
        call    putHex          ;display number in hexadecimal
;Display a LARGE number in hexadecimal
        mov     eax, number-to-be-displayed
        call    putDHex                 ;display number in decimal
```

Inputting Numbers

To input a decimal number from the Keyboard:

```
call GetDec     ;Typed number is now in ax (in binary)
call GetDDec    ;Typed number is now in eax (in binary)
```

Note that **GetDec** doesn't display any prompt, so you probably have to use **_PutStr** to inform the user that the program is waiting for an input. A possible snippet might be:

```
FirstNumber     DW      ?
Prompt          DB      'Type in first integer: $'   ; Note: No CR-LF
                ...
                _PutStr  Prompt
                call    GetDec
                mov     FirstNumber, ax
```

Since the CR-LF is omitted in Prompt, the typed input will be beside the prompt. For example (user typing showed in bold):

Type in first integer: **1234** ↵
The hex equivalent is: 04D2H

Assembling the program using GetDec, PutDec, etc… is same as before, but when you are linking the program, the linker must be told where to find these procedures. The appropriate command is

link or tlink *MyProgfilename* , , ,util;

Note: Exactly three commas are needed. The two other parameters mean that default value will be used. If **util** is used on its own it is implied that the file **UTIL.LIB** is in the current directory. If for instance, it is in the directory **c:\masm615**, then you will have to enter:

link or tlink *MyProgfilename*, , ,c:\masm615\util;

With **masm** you can assemble and link in one step:

ml ***MyProgfilename*.asm util.lib**

Notice that the extensions **.asm** and **.lib** are essential to tell **ml** what to do with the various files. If **util.lib** is not in the current directory, its actual directory must be specified. *Note that the **ml** command does not have commas.*

Example Program: DecToHex.asm

Write an assembly language program **DecToHex** that reads in a decimal number from the keyboard and displays its representation in hex.

The **.CODE** part for a straightforward program to do the conversion could be:

```
            .CODE
            EXTRN  GetDec : NEAR, PutHex : NEAR
Dec2Hex     PROC
            _Begin
            call   GetDec        ;ax = decimal number from keyboard
            call   PutHex        ;display contents of ax in hex
            _Exit  0             ;return to DOS
Dec2Hex     ENDP
            END          Dec2Hex
```

This program will work but it is not user-friendly. It should really contain a prompt telling the user what to enter and a message describing the output.
But if we write:

```
Prompt      DB           'Enter decimal number: $'
Message     DB           'The hex equivalent is: $'
            ...
            _PutStr Prompt
            call   GetDec
            _PutStr Message      WRONG!!!
            call   PutHex
```

The hex value will be garbage. **Why?**
The reason is that **_PutStr** destroys **ah**, and thus **ax** as well.

Solution: We need to save **ax** before the **second _PutStr** call and restore it after. The easiest way to save and restore a register is to set up a **word (16-bit) variable** in which to save it.

```
Prompt              DB      'Enter decimal number: $'
Message             DB      'The hex equivalent is: $'
SaveAX      DW      ?
                    ...
        _PutStr Prompt
        call        GetDec
        mov         SaveAX, ax          ;;save ax in variable SaveAX
        _PutStr     Message
        mov         ax, SaveAX          ;;Restore value of ax from SaveAX
        call        PutHex
```

Note: We could have used another register instead of **SaveAX**, say **bx** or **cx** but not **dx**. ***We have to be sure that the registers will be unaltered by DOS calls.***

To be on the safe side, store data in variables instead of registers; unless that you are sure that the registers' data will not be changed.

The complete program is below. A trailing 'H' has been added as **putHex** does not add it.

;; **DecToHex.asm** – input of a decimal number from keyboard and displays its hex equivalent

```
INCLUDE     PCMAC.INC
            .MODELSMALL
            .586
            .STACK  100h

            .DATA
Prompt      DB      'Enter decimal number: $'
Message     DB      'The hex equivalent is: $'
SaveAX      DW      ?

            .CODE
            EXTRN   GetDec : NEAR, PutHex : NEAR
Dec2Hex     PROC
            _Begin
            _PutStr Prompt
            call    GetDec
            mov     SaveAX, ax
            _PutStr Message
            mov     ax, SaveAX
            call    PutHex
            _PutCh  'H', 13, 10     ;Display trailing 'H' for hex and CR-LF
            _Exit   0               ;return to DOS
Dec2Hex     ENDP
            END     Dec2Hex
```