Library of common functions: emu8086.inc

8086 assembler tutorial for beginners (part 5)

To make programming easier there are some common functions that can be included in your program. To make your program use functions defined in other file you should use the **INCLUDE** directive followed by a file name. Compiler automatically searches for the file in the same folder where the source file is located, and if it cannot find the file there - it searches in **Inc** folder.

Currently you may not be able to fully understand the contents of the **emu8086.inc** (located in **Inc** folder), but it's OK, since you only need to understand what it can do.

To use any of the functions in **emu8086.inc** you should have the following line in the beginning of your source file:

```
include 'emu8086.inc'
```

---

emu8086.inc defines the following **macros**:

- **PRINT string** - macro with 1 parameter, prints out a string.
- **PRINTN string** - macro with 1 parameter, prints out a string. The same as PRINT but automatically adds "carriage return" at the end of the string.
- **PUTC char** - macro with 1 parameter, prints out an ASCII char at current cursor position.
- **GOTOXY col, row** - macro with 2 parameters, sets cursor position.
- **CURSOROFF** - turns off the text cursor.
- **CURSORON** - turns on the text cursor.

To use any of the above macros simply type its name somewhere in your code, and if required parameters, for example:

```
include emu8086.inc

ORG    100h

PRINT 'Hello World!'

GOTOXY 10, 5
```

```
PUTC 65           ; 65 - is an ASCII code for 'A'
PUTC 'B'

RET               ; return to operating system.
END               ; directive to stop the compiler.
```

When compiler process your source code it searches the **emu8086.inc** file for declarations of the macros and replaces the macro names with real code. Generally macros are relatively small parts of code, frequent use of a macro may make your executable too big (procedures are better for size optimization).

---

emu8086.inc also defines the following **procedures**:

- **PRINT_NUM** - procedure that prints a signed number in **AX** register. To use it declare: **DEFINE_PRINT_NUM** and **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **PRINT_NUM_UNS** - procedure that prints out an unsigned number in **AX** register. To use it declare: **DEFINE_PRINT_NUM_UNS** before **END** directive.
- **GET_STRING** - procedure to get a null terminated string from a user, the received string is written to buffer at **DS:DI**, buffer size should be in **DX**. Procedure stops the input when 'Enter' is pressed. To use it declare: **DEFINE_GET_STRING** before **END** directive.
- **PRINT_STRING** - procedure to print a null terminated string at current cursor position, receives address of string in **DS:SI** register. To use it declare: **DEFINE_PRINT_STRING** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before the **END** directive), and then use **CALL** instruction followed by a procedure name. For example:

```
; demonstrate get_string and print_string
;----------------------------------------
include 'emu8086.inc'
ORG    100h

LEA    SI, msg1       ; set up pointer (SI) to msg
                      ; to ask for the number
CALL   print_string   ; print message that SI points to

LEA    DI, buffer     ; set up pointer (DI) to input buffer
MOV    DX, bufSize    ; set size of buffer
CALL   get_string     ; get name & put in buffer

LEA    SI, newln      ; point at CR/LF / Hello message
CALL   print_string   ; print message that SI points to
```

```
RET                   ; return to operating system.

; data
msg1   DB "Enter your name: ", 0
newln  DB 13, 10
       DB "Hello, "
buffer DB 20 DUP (0)  ; input buffer for get_string
bufSize = $-buffer    ; calculates size of buffer

DEFINE_GET_STRING
DEFINE_PRINT_STRING
END                   ; directive to stop the compiler.
```

- **CLEAR_SCREEN** - procedure to clear the screen, (done by scrolling entire screen window), and set cursor position to top of it. To use it declare: **DEFINE_CLEAR_SCREEN** before **END** directive.

- **PTHIS** - procedure to print a null terminated string at current cursor position (just as PRINT_STRING). The ZERO TERMINATED string should be defined just after the CALL. For example:

  ```
  CALL PTHIS
  db 'Hello World!', 0
  ```

  Address of string is stored in the Stack as return address. Procedure updates value in the Stack to make return after string definition. To use it declare: **DEFINE_PTHIS** before **END** directive.

- **SCAN_NUM** - procedure that gets the multi-digit SIGNED number from the keyboard, and stores the result in **CX** register. To use it declare: **DEFINE_SCAN_NUM** before **END** directive.

To use any of the above procedures you should first declare the function in the bottom of your file (but before the **END** directive), and then use **CALL** instruction followed by a procedure name. For example:

```
; demonstrate scan_num, print_num, pthis
;----------------------------------------
include 'emu8086.inc'
ORG    100h

LEA    SI, msg1       ; ask for the number
CALL   print_string   ;
CALL   scan_num       ; get number in CX.

MOV    AX, CX         ; copy the number to AX.

; print the following string:
CALL   pthis
DB  13, 10, 'You have entered: ', 0

CALL   print_num      ; print number in AX.

RET                   ; return to operating system.
```

```
; data
msg1   DB  'Enter the number: ', 0

; macros to define procs
DEFINE_SCAN_NUM
DEFINE_PRINT_STRING
DEFINE_PRINT_NUM
DEFINE_PRINT_NUM_UNS  ; required for print_num.
DEFINE_PTHIS

END                   ; directive to stop the compiler.
```

First, the compiler processes the declarations (these are just regular the macros that are expanded to procedures). When compiler gets to **CALL** instruction it replaces the procedure name with the address of the code where the procedure is declared. When **CALL** instruction is executed control is transferred to procedure. This is quite useful, since even if you call the same procedure 100 times in your code you will still have relatively small executable size. Seems complicated, isn't it? That's ok, with the time you will learn more, currently it's required that you understand the basic principle.

<<< previous part <<<   **>>> Next Part >>>**