# ECE 3210 LABORATORY 6
# INTERFACING ASSEMBLY LANGUAGE WITH C LANGUAGE

## OBJECTIVE

The interfacing between C language and assembly language is learned in this lab assignment. A program in C is given, so you are asked to write a program in assembly for linking with the provided C program. The approach of interfacing subroutines with parameter passing is the key to perform this work successfully. After finishing this lab, you should be familiar with the procedure of C called by assembler or assembler called by C.

## BACKGROUND

1. Although it is common to call assembly functions from C to perform a specialized task, such as hardware access or performance timing minimization, etc., you may occasionally want to call C functions from assembler. As it turns out, it is actually easier to call a C function from an assembly function than the reverse, since no stack-frame handling on the part of the assembler code is required. One case in which you may want to call a C function from assembler is when you need to perform complex calculations. This is especially true when mixed integer and floating-point calculations are involved.

2. Upon return from a function, the parameters that were pushed on the stack are still there, but are no longer of any use. Consequently, immediately following each function call, Turbo C adjusts the stack pointer back to the value it contained before t he parameters were pushed, thereby discarding the parameters. Assembly functions can access parameters passed on the stack relative to the BP register after loading the BP with SP. However, you must first preserve BP, since the calling C code expects you to return with BP unchanged.

3. Turbo C passes parameters to functions on the stack. Before calling a function, Turbo C first pushes the parameters to that function on the stack, starting with the rightmost parameter and ending with the leftmost parameter.

4. As far as Turbo C is concerned, C-callable assembly functions can do anything they please, as long as they preserve the following registers: BP, SP, CS, and SS. AX, BX, CX, DX, ES, and the flags can be changed in any way. It is a good practice to always preserve SI and DI in your C-callable assembly functions.

5. In general, 8 and 16-bit values are returned in AX, and 32-bit values are returned in DX:AX. Floating-point values are returned in ST(0), which is the 8087's top-of-stack (TOS) register or in the 8087 emulator's TOS register if the floating-point emulator is being used.

6. Normally, Turbo C expects all the external labels to start with an underscore character "_". You must be sure that all assembler references to C functions and variables begin

with underscores, and you must begin all assembler functions and variables that are made PUBLIC and referenced by C code with underscore. Also remember to put EXTRN directives in the assembly function for far symbols referenced either outside all segments or inside the correct segment.

7. Turbo Assembler is normally insensitive to case when handling symbolic names. Since C is case sensitive, it is desirable to have Turbo Assembler be case sensitive, at least for those symbols that are shared between assembler and C.

8. The dot and cross product in three dimensions are vector operations common in physics. The dot product produces a scalar norm of two vectors, whereas the cross product produces a vector orthogonal to the original two. Below are the formulas for calculation of the dot and cross product, respectively.

$A \bullet B = A_xB_x + A_yB_y + A_zB_z$

$A \times B = i\,(A_yB_z - A_zB_y) + j\,(A_zB_x - A_xB_z) + k\,(A_xB_y - A_yB_x)$

**PRELAB**

1. Explain the input/output of the **matrix.c** file.
2. In modular programming we need to allow communication between modules.
    a. Explain the use of PUBLIC and EXTERN directives
    b. Can one of them be used without the other? Explain.
3. Refer to **Lab6_ref.pdf** to answer the following:
    a. What does INT 03 do?
    b. Do we need to use the line " MOV BP, SP "? Explain your answer
    c. What will happen if we use " MOV AX, [SP+4]?
    d. In page 2, what is the value of vector **a**? Where in the memory it is stored (code, data, stack), and at what address?

**REQUIREMENTS**

1. The matrix.c file is provided in the web page and cannot be changed. You are asked to write a module in assembly the implements the dot and outer product as separate functions. Both take pointers to the vector structure, which is simply three 16-bit signed integers: x, y, and z respectively, and the lvector structure which is a 32 bit version. The first two pointers correspond to the 16-bit operands, and the third points two the 32 bit result. The prototype of these assembly functions in C looks as follows:

```
extern void outP(vector* a, vector* b, lvector* result) ;

extern void crossP(vector*a, vector* b, long int* result) ;
```

The module calculates the dot product of a and b, storing the 32bit scalar product in result. The module also calculates the cross product, storing the 32bit three dimensional vector in result. The operands should not be modified.

2. When the C program, matrix.c, calls the assembly routine, it passes the address of the variable data as a variable. Since we have a small model, the procedures are near ones. This means that only the offset of the address for variable data gets pushed on the stack. Upon entry to the assembly routine, the top of the stack looks like the following:

Top of Stack :Return Address
Top of Stack +2: Address of vector a
Top of Stack +4: Address of vector b
Top of Stack +6: Address of result

## LABORATORY

1. The format of the assembler function is shown below:

```
.MODEL SMALL
.DATA
  :
  :
.CODE
PUBLIC _crossP
_crossP PROC
  :
  :
  :
RET
_crossP ENDP
END
```

2. To create the executable program **matrix.exe** from **matrix.c** and **mmath.asm**, enter the command line:

**tcc -M matrix.c -v mmath.asm**

where the parameter -M is to generate link MAP file. It's convenient for you to locate the offset of the function for debugging. By the way, you don't need to use TASM to assemble your assembly source code. TCC takes care of everything.

3. To debug the program, you can use Turbo Debugger. You can insert INT 03h (Breakpoint) where you want the program to break.

Here is an example of how the output should look like:

```
C:\>matrix
Enter the first 3 dimensional vector below
enter x component: 6

enter y component: 2

enter z component: 12
Enter the second 3 dimensional vector below
enter x component: -18
```

```
enter y component: -4

enter z component: 5
Cross Product:
x:58    y:-246  z:12

Dot Product: : -56
Continue y/n [n]: n
C:\>
```

***Note:*** *the bold characters indicate the user inputs.*