16 Mating Assembly with C

"Do to others what you want them to do to you."

Nothing can beat the efficiency of Assembly language. A good optimizing C compiler will convert C file to a better assembly code. But a good human Assembly programmer can write much more tight and efficient code. If you are such an efficient-superb Assembly programmer, fortunately there is a way to link those assembly codes with C and so you can improve your program.

16.1 Inline Assembly

You can write Assembly code inside a C file. That is called as Inline Assembly. In TC++3.0 Inline assembly is being assembled by BASM (Built-in inline Assembler). You don't need TASM. If you use `#pragma inline`, inline codes get assembled with TASM. If you use x386 instructions in inline assembly, BASM cannot assemble those codes. In such a case you must use TASM and for that you should use `#pragma inline`.

16.1.1 Example 1

Let's see an example to print message "A to Z of C" with inline assembly.

```
int main( void )
{
   char *msg = "A to Z of C \r\n$";  /* $ is the null terminator
                                        in assembly */
   asm {
        MOV AH, 9;
          MOV DX, msg;
          INT 21H;
       }
   return(0);
} /*--main( )-----*/
```

Here we have used interrupts to print message. We can see more about interrupt programming later.

16.1.2 Example 2

We can also use inline assembly in functions. Anything that is present in AX register will be returned.

Let's see a program to add two integers.

```
/* main program */
int main( void )
{
   printf( "5+100 = %Ld\n", Add( 5, 100 ) );
   return(0);
} /*--main( )-------*/
```

Now we have to write the function `Add( )` with inline assembly.

```
int Add( int x, int y )
{
   asm {
        MOV AX, x;
        MOV BX, y
        ADD AX, BX;
       }
  /* return(_AX);  can be used to shut off warning */
} /*--Add( )-------*/
```

So the result in AX gets returned automatically. But here you will get a warning. If you are allergic to warning, you can shut it off by adding `return( _AX  );` in the last line.

Let's see another efficient version of `Add( )`.

```
int Add ( int _AX, int _BX )
{
   asm ADD AX, BX;
} /*--Add( )-------*/
```

If you want to return long values, you can use

```
long Add( int x, int y )
{
   asm{
       MOV DX, 0;
       MOV DX, x;
       ADD AX, y;  /* low byte in AX */
       ADC DX, 0;  /* high byte in DX */
      }
} /*--Add( )-------*/
```

The result in AX(upper word), DX(lower word) gets returned as long. Here you must *not* use `return( _AX);` to shut off warning!

16.1.3 Usual Errors

Most of the time you don't need TASM because the built-in BASM is sufficient enough. In case if you use x386 instructions, you have to invoke TASM with `#pragma inline`. You will get error when you don't have TASM assembler. One solution for this error is to buy TASM from Borland for about $130 (TASM is not yet available for free). Another solution is to create a separate and a pure (i.e., without C) assembly file and assemble with the free assembler like NASM, MASM, etc. Then you have to link that OBJ file with C (This technique of calling Assembly routine from C is discussed in the next section).

16.2 Calling Assembly routines from C

Believe it or not, all the standard library functions are written in Assembly (not in C!!) by Borland for efficiency. Then you might be asking me how is it possible to call such a routine from C. Yes, it is possible. The idea is you can link any portable OBJ and LIB files. Thus the standard library functions that are available as LIB and OBJ (browse to your TC folder and check!!) are being linked by the linker with C files in 'linking phase'.

16.2.1 C's calling convention

Before getting deeper on this subject it is necessary to know about the convention of C language. In high level language whenever a function is being called, the parameters are pushed into the stack so that the parameters be passed to that routine. For example, if we call a function `Add(7,70)`, the parameters 7 and 70 are pushed into the stack. The order in which the parameters are pushed varies from language to language. In C language the parameters are pushed in the reverse order (i.e., 70 first, then 7). Also C passes the parameters by value rather than by reference, unless we have used pointers.

Calling convention of high level language		
	Parameter passing	**Destination**
C	by value	Reverse Order
Pascal	by value	In the given order
FORTRAN	by reference	In the given order

We can also set our TC IDE to use Pascal calling function by `OPTION > COMPILER > PASCAL`. in the command line `TCC -p`. When you use such Pascal calling conventions, you must explicitly declare `main( )` with `cdecl` as

```
int cdecl main( void )
```

Note

As the Pascal calling convention ensures 'In Order' pushing, it produces tight & efficient code. However it is a good practice to stick onto the C's standard calling convention.

16.2.2 C's naming convention

When you declare an identifier, Turbo C automatically joins an underscore in front of the identifier before saving that identifier in that object module. However, Turbo C treats Pascal type identifiers (those modules with `pascal` keyword) differently. i.e., they use uppercase and are not prefixed with underscore. Turbo C automatically joins an underscore in front of the function name too.

16.2.3 Example 1

With the above enough theory let's see a real example of how to link the assembly routines with C. Please note that in assembly the comment line starts with semicolon (;).

```
; File name: Hello1.asm
.MODEL small
.DATA
     msg DB "Hello!$"
.CODE
     PUBLIC _PrintHello       ;     Function Name
_PrintHello PROC NEAR
     MOV AH, 9
     MOV DX, OFFSET msg
     INT 21h
     RET
_PrintHello ENDP
END
```

Here you might have noticed that we have prefixed underscore (_) with the name of the function. That is because of the C's naming convention as discussed in the previous section. You have to note that we are mating two different language i.e. C and Assembly. As we discussed, when we compile a C file to OBJ file all the function names and identifiers are automatically prefixed with underscore (_) by the compiler. So if we don't put up an underscore (_) here in Assembly, we cannot link these files. If you find it odd to use an underscore (_) in front of function name, then there is another way of declaring function i.e. to use 'C' keywords with assembly directive as:

```
;File name: Hello2.asm
.MODEL small, C   ;'C' used to set the assembly to C
                  ; calling & naming convention
.DATA
     msg DB "Hello!$"
.CODE
```

```
     PUBLIC PrintHello
PrintHello PROC NEAR
     MOV AH, 9
     MOV DX, OFFSET msg
     INT 21h
     RET
PrintHello ENDP
END
```

The 'C' keyword sets the assembler to use C calling convention and it automatically prefixes underscore(_) with all procedures that are declared as EXTERN or PUBLIC. Here we find that Hello2.asm "*looks better*" than Hello1.asm! So let's use Hello2.asm.

The next step is to assemble the Hello2.asm to OBJ file. When you assemble, you must assemble it with the case sensitive switch on. The assembler makes all PUBLIC labels into capital letters by default, unless we use case sensitive switch -mx. Case sensitive is important, because C language is case sensitive and we need "PrintHello" to be case sensitive. We can assemble the Hello2.asm as:

```
C:\WAR>TASM -mx Hello2.asm
```

Now you will get Hello2.OBJ which contains PrintfHello procedure.

Note

You can even assemble the Hello2.asm from IDE by choosing ≡>Turbo Assembler

Note

If you don't have TASM, you can use the available assemblers such as MASM, NASM etc. For the details regarding the switches, see your assembler's documentation.

Next we have to write a C program that uses `PrintHello( )` function.

```
/*  Chkasm1.c  */
extern PrintHello( void ); /* PrintHello is written in assembly
                           available in Hello2.asm */
int main( void )
{
   PrintHello( );
   return (0);
} /*--main( )-----*/
```

Now we have to compile chkasm1.c and link Hello2.obj in the same time as:

```
C:\WAR> tcc chkasm1.c Hello2.obj
```

Now you will get `chkasm1.exe` that you can run it under DOS.

Note

To compile chkasm1.c and link Hello2.obj, you can also use project file instead of command line compiler tcc.

16.2.4 Example 2

```
; File name: Addnum.asm
.MODEL small, C
.CODE
    PUBLIC Addnum
Addnum PROC NEAR USES BX, x: WORD, y: WORD
    MOV AX, x
    ADD AX, y
    RET
Addnum ENDP
END
```

Assemble as : `c:\WAR>TASM -mx Addnum`

```
/* Chkasm2.c */
extern Addnum( int x, int y ); /* Addnum is written in
                                  Addnum.asm */
int main( void )
{
   printf( "5+100 = %d \n", Addnum( 5, 100 ) );
   return(0);
} /*--main( )---*/
```

Compile and link as : `c:\WAR>tcc chkasm2.c addnum.obj`

16.3 Creating library file out of assembly language module

Creating library file out of assembly language module is the easiest one. We can add any number of modules with the library file. For that you can use TLIB. For example to create a library file newlib.lib which contains our `PrintHello( )` and `Add( )` functions we can use,

```
C:\WAR>TLIB NEWLIB.LIB + Hello2.OBJ
```

Now the newlib.lib file contains only the `PrintHello( )` function.

```
C:\WAR>TLIB NEWLIB.LIB + addnum.obj
```

Now the newlib.lib file contains both `PrintHello( )` and `Addnum( )` function.

If you feel that `newlib.lib` should not contain `PrintHello( )` function, you can even remove the function with the help of '-' switch as:

```
C:\WAR>TLIB NEWLIB.LIB - Hello2.obj
```

For more information on the switch of TLIB, see the Turbo C documentation.