INLINE ASSEMBLY LANGUAGE PROGRAMS IN C/C++

SUBMITTED BY,

Introduction

First of all, what does term "inline" mean?

- Generally the inline term is used to instruct the compiler to insert the code of a function into the code of its caller at the point where the actual call is made. **Such functions are called "inline functions".** The benefit of inlining is that it reduces function-call overhead.
- Now, it's easier to guess about **inline assembly**. It is just a **set of assembly instructions** written as inline functions.

Introduction

- Assembly language serves many purposes, such as **improving program speed, reducing memory needs, and controlling hardware.**
- We can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps.
- We can mix the assembly statements within C/C++ programs using **keyword asm**

Inline Assembler Overview

- The inline assembler lets us embed assembly-language instructions in our C and C++ source programs **without extra assembly and link steps.**
- Inline assembly code can use any C or C++ variable or function name that is in scope.
- The **asm** keyword **invokes the inline assembler** and **can appear wherever a C or C++ statement is legal.** It cannot appear by itself.
- It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces.

Advantages of Inline Assembly

- The inline assembler doesn't require separate assembly and link steps, it is **more convenient than a separate assembler.**
- Inline assembly code can use any C variable or function name that is in scope, so it is **easy to integrate** it with our program's C code.
- Because the assembly code can be mixed inline with C or C++ statements, **it can do tasks that are cumbersome or impossible in C or C++.**

Advantages of Inline Assembly

The uses of inline assembly include:

- Writing functions in assembly language.
- Spot-optimizing speed-critical sections of code.
- Making direct hardware access for device drivers.
- Writing prolog and epilog code for "naked" calls.

asm

- The **asm** keyword invokes the inline assembler and can appear wherever a C or C++ statement is legal. It cannot appear by itself.
- It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces.

- **Grammar**

asm-statement:

asm *assembly-instruction* **;**$_{opt}$

asm { ***assembly-instruction-list* **} **;**$_{opt}$

assembly-instruction-list:

assembly-instruction **;**$_{opt}$

assembly-instruction **;** *assembly-instruction-list* **;**$_{opt}$

asm

- The following code fragment is a simple **asm** block enclosed in braces:

```
asm {
        mov al, 2
        mov dx, 0xD007
        out dx, al
         }
```

asm

- Alternatively, you can put **asm** in front of each assembly instruction:

```
asm mov al, 2
asm mov dx, 0xD007
asm out dx, al
```

- Because the **asm** keyword is a statement separator, you can also put assembly instructions on the same line:

```
asm mov al, 2 asm mov dx, 0xD007 asm out dx, al
```

asm

- All three examples generate the same code, but the first style (enclosing the **asm** block in braces) has some advantages.
- The braces clearly separate assembly code from C or C++ code and avoid needless repetition of the **asm** keyword.
- Braces can also prevent ambiguities. If you want to put a C or C++ statement on the same line as an **asm** block, you must enclose the block in braces. Without the braces, the compiler cannot tell where assembly code stops and C or C++ statements begin.

Using C or C++ in asm Blocks

Because inline assembly instructions can be mixed with C or C++ statements, they can refer to C or C++ variables by name and use many other elements of those languages.

An asm block can use the following language elements:

- Symbols, including labels and variable and function names
- Constants, including symbolic constants and **enum** members
- Macros and preprocessor directives
- Comments (both ***/* */*** and ***//***)
- Type names (wherever a MASM type would be legal)
- **typedef** names, generally used with operators such as **PTR** and **TYPE** or to specify structure or union members

Jumping to Labels in Inline Assembly

- Like an ordinary C or C++ label, a label in an asm block has scope throughout the function in which it is defined (not only in the block). **Both assembly instructions and goto statements can jump** to **labels inside or outside the asm block**.
- Labels defined in asm blocks are not case sensitive; both goto statements and assembly instructions can refer to those labels without regard to case. C and C++ labels are case sensitive only when used by goto statements. Assembly instructions can jump to a C or C++ label without regard to case.

Jumping to Labels in Inline Assembly

- **Example:**

```
Void func( void )
        {
                goto C_Dest;
                goto A_Dest;
        asm {
                jmp C_Dest ;
                jmp A_Dest ;
                a_dest: ;          asm label
              }
                C_Dest:            /* C label */
                return;
         }
        int main()
         { }
```

Example


```
#include<stdio.h>
#include<conio.h>
void main()
{
        char a[5]="ODD$";
        char b[5]="EVEN$";
net:
                asm{
                mov cl,al
                mov bl,13
                mov ah,0x01
                int 0x21
                cmp bl,al
                jnz net
                mov al,cl
                sub al,0x30
                shr al,1
JNC next
lea dx,a
mov ah,09h
int 21h
jmp er
}
next:
                asm
{
                lea dx,b
                mov ah,09h
                int 21h
                jmp er
}
er:
                getch();
}
```

Calling C Functions in Inline Assembly

- An __**asm** block can call C functions, including C library routines. The following example calls the **printf** library routine:

Example:

```
char format[]=" %s %s\n";
char hello[]="hello";
char world[]="world";
void main()
        {
                asm{
                mov ax,offset world
                push ax
                mov ax,offset hello
                push ax
                mov ax,offset format
                push ax
                call printf
                pop bx
                pop bx
                pop bx
        }
```

Calling C Functions in Inline Assembly

- Because function arguments are passed on the stack, you simply push the needed arguments — string pointers, in the previous example — before calling the function. The arguments are pushed in reverse order, so they come off the stack in the desired order. To emulate the C statement

printf(format, hello, world);

- the example pushes pointers to world, hello, and format, in that order, and then calls **printf**.

Calling C++ Functions in Inline Assembly

- An **asm** block can call only global C++ functions that are not overloaded.
- If we call an overloaded global C++ function or a C++ member function, the compiler issues an error.
- We can also call any functions declared with **extern "C"** linkage.
- This allows an **asm** block within a C++ program to call the C library functions, because all the standard header files declare the library functions to have **extern "C"** linkage.

Defining asm Blocks as C Macros

- C macros offer a convenient way to insert assembly code into your source code, but they demand extra care because a macro expands into a single logical line. To create trouble-free macros, follow these rules:
  - **Enclose the asm block in braces.**
  - **Put the asm keyword in front of each assembly instruction.**
  - **Use old-style C comments (/* comment */) instead of assembly-style comments (; comment) or single-line C comments (// comment).**

Defining asm Blocks as C Macros

- To illustrate, the following example defines a simple macro:

```
#define PORTIO asm
                {
                asm mov al, 2
                asm mov dx, 0xD007
                asm out dx, al
                 }
```

Optimizing Inline Assembly

- The presence of an **asm** block in a function affects optimization in several ways.
  - **First, the compiler doesn't try to optimize the asm block itself. What you write in assembly language is exactly what you get.**
  - **Second, the presence of an asm block affects register variable storage. The compiler avoids enregistering variables across an asm block if the register's contents would be changed by the asm block.**
  - **Finally, some other function-wide optimizations will be affected by the inclusion of assembly language in a function.**

THANK YOU