Advantages of Inline Assembly

Because 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 your 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++.
The uses of inline assembly include:

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

Inline assembly is a special-purpose tool. If you plan to port an application to different machines, you'll probably want to place machine-specific code in a separate module. Because the inline assembler doesn't support all of Microsoft Macro Assembler's (MASM) macro and data directives, you may find it more convenient to use MASM for such modules.

The __asm Keyword

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. The term "**__asm** block" here refers to any instruction or group of instructions, whether or not in braces.
The following code fragment is a simple **__asm** block enclosed in braces:

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

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

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

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 al, dx
```

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. Finally, because the text in braces has the same format as ordinary MASM text, you can easily cut and paste text from existing MASM source files.
Unlike braces in C and C++, the braces enclosing an **__asm** block don't affect variable scope. You can also nest **__asm** blocks; nesting does not affect variable scope.

Instruction Set for Inline Assembly

The inline assembler supports the full instruction set of the Intel 486 processor. Additional instructions supported by the target processor can be created with the _emit Pseudoinstruction.

The **_emit** pseudoinstruction is similar to the **DB** directive of MASM. You use **_emit** to define a single immediate byte at the current location in the current text segment. However, **_emit** can define only one byte at a time, and it can only define bytes in the text segment. It uses the same syntax as the **INT** instruction.
The following fragment places the given bytes into the code:

```
#define randasm __asm _emit 0x4A __asm _emit 0x43 __asm _emit 0x4B
 .
 .
 .
__asm {
     randasm
     }
```

MASM Expressions in Inline Assembly

Inline assembly code can use any MASM expression, which is any combination of operands and operators that evaluates to a single value or address.

Data Directives and Operators in Inline Assembly

Although an **__asm** block can reference C or C++ data types and objects, it cannot define data objects with MASM directives or operators. Specifically, you cannot use the definition directives **DB**, **DW**, **DD**, **DQ**, **DT**, and **DF**, or the operators **DUP** or **THIS**. MASM structures and records are also unavailable. The inline assembler doesn't accept the directives **STRUC**, **RECORD**, **WIDTH**, or **MASK**.

Although the inline assembler doesn't support most MASM directives, it does support **EVEN** and **ALIGN**. These directives put **NOP** (no operation) instructions in the assembly code as needed to align labels to specific boundaries. This makes instruction-fetch operations more efficient for some processors.

MASM Macro Directives in Inline Assembly

The inline assembler is not a macro assembler. You cannot use MASM macro directives (**MACRO**, **REPT**, **IRC**, **IRP**, and **ENDM**) or macro operators (**<>**, **!**, **&**, **%**, and **.TYPE**). An **__asm** block can use C preprocessor directives, however.

Segment References in Inline Assembly

You must refer to segments by register rather than by name (the segment name **_TEXT** is invalid, for instance). Segment overrides must use the register explicitly, as in ES:[BX].

Type and Variable Sizes in Inline Assembly

The **LENGTH**, **SIZE**, and **TYPE** operators have a limited meaning in inline assembly. They cannot be used at all with the **DUP** operator (because you cannot define data with MASM directives or operators). But you can use them to find the size of C or C++ variables or types:

- The **LENGTH** operator can return the number of elements in an array. It returns the value 1 for non-array variables.

- The **SIZE** operator can return the size of a C or C++ variable. A variable's size is the product of its **LENGTH** and **TYPE**.

- The **TYPE** operator can return the size of a C or C++ type or variable. If the variable is an array, **TYPE** returns the size of a single element of the array.

For example, if your program has an 8-element **int** array,

```
int arr[8];
```

the following C and assembly expressions yield the size of `arr` and its elements.

__asm	**C**	**Size**
LENGTH arr	**sizeof**(arr)/**sizeof**(arr[0])	8
SIZE arr	**sizeof**(arr)	16
TYPE arr	**sizeof**(arr[0])	2

Assembly-Language Comments

Instructions in an **__asm** block can use assembly-language comments:

```
__asm mov ax, offset buff ; Load address of buff
```

Because C macros expand into a single logical line, avoid using assembly-language comments in macros. (SeeDefining __asm Blocks as C Macros.) An **__asm** block can also contain C-style comments; for more information, see Using C or C++ in __asm Blocks.

Debugging and Listings for Inline Assembly

Programs containing inline assembly code can be debugged with a source-level debugger if you compile with the /Zi option.

Within the debugger, you can set breakpoints on both C or C++ and assembly-language lines. If you enable mixed assembly and source mode, you can display both the source and disassembled form of the assembly code.

Note that putting multiple assembly instructions or source language statements on one line can hamper debugging. In source mode, you can use the debugger to set breakpoints on a single line but not on individual statements on the same line. The same principle applies to an **__asm** block defined as a C macro, which expands to a single logical line.

If you create a mixed source and assembly listing with the /FAs compiler option, the listing contains both the source and assembly forms of each assembly-language line. Macros are not expanded in listings, but they are expanded during compilation

Intel's MMX Instruction Set

The Visual C++ compiler allows you to use Intel's MMX (multimedia extension) instruction set in the inline assembler. The MMX instructions are also supported by the debugger disassembly. The MMX registers are not supported in the debugger register window. The compiler generates a warning message if the function contains MMX instructions, but does not have an EMMS instruction to empty the multimedia state. For more information, see the Intel Web site.

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

Within an **__asm** block, you can specify integer constants with either C notation or assembler radix notation (0x100 and 100h are equivalent, for example). This allows you to define (using **#define**) a constant in C and then use it in both C or C++ and assembly portions of the program. You can also specify constants in octal by preceding them with a 0. For example, 0777 specifies an octal constant.

Using Operators in __asm Blocks

An **__asm** block cannot use C or C++ specific operators, such as the << operator. However, operators shared by C and MASM, such as the * operator, are interpreted as assembly-language operators. For instance, outside an **__asm** block, square brackets (**[ ]**) are interpreted as enclosing array subscripts, which C automatically scales to the size of an element in the array. Inside an **__asm** block, they are seen as the MASM index operator, which yields an unscaled byte offset from any data object or label (not just an array). The following code illustrates the difference:

```
int array[10];

__asm mov array[6], bx ;  Store BX at array+6 (not scaled)

array[6] = 0;          /* Store 0 at array+12 (scaled) */
```

The first reference to `array` is not scaled, but the second is. Note that you can use the **TYPE** operator to achieve scaling based on a constant. For example, the following statements are equivalent:

```
__asm mov array[6 * TYPE int], 0 ; Store 0 at array + 12

array[6] = 0;                    /* Store 0 at array + 12 */
```

Using C or C++ Symbols in __asm Blocks

An **__asm** block can refer to any C or C++ symbol in scope where the block appears. (C and C++ symbols are variable names, function names, and labels; that is, names that aren't symbolic constants or **enum** members. You cannot call C++ member functions.)

A few restrictions apply to the use of C and C++ symbols:

- Each assembly-language statement can contain only one C or C++ symbol. Multiple symbols can appear in the same assembly instruction only with **LENGTH**, **TYPE**, and **SIZE** expressions.
- Functions referenced in an **__asm** block must be declared (prototyped) earlier in the program. Otherwise, the compiler cannot distinguish between function names and labels in the **__asm** block.
- An **__asm** block cannot use any C or C++ symbols with the same spelling as MASM reserved words (regardless of case). MASM reserved words include instruction names such as **PUSH** and register names such as SI.
- Structure and union tags are not recognized in **__asm** blocks.

Accessing C or C++ Data in __asm Blocks

A great convenience of inline assembly is the ability to refer to C or C++ variables by name. An **__asm** block can refer to any symbols, including variable names, that are in scope where the block appears. For instance, if the C variable `var` is in scope, the instruction

```
__asm mov eax, var
```

stores the value of `var` in EAX.

If a class, structure, or union member has a unique name, an **__asm** block can refer to it using only the member name, without specifying the variable or **typedef** name before the period (**.**) operator. If the member name is not unique, however, you must place a variable or **typedef** name immediately before the period operator. For example, the following structure types share `same_name` as their member name:

```
struct first_type
{
   char *weasel;
   int same_name;
};

struct second_type
{
   int wonton;
   long same_name;
};
```

If you declare variables with the types

```
struct first_type hal;
struct second_type oat;
```

all references to the member `same_name` must use the variable name because `same_name` is not unique. But the member `weasel` has a unique name, so you can refer to it using only its member name:

```
__asm
{
   mov ebx, OFFSET hal
   mov ecx, [ebx]hal.same_name ; Must use 'hal'
   mov esi, [ebx].weasel       ; Can omit 'hal'
}
```

Note that omitting the variable name is merely a coding convenience. The same assembly instructions are generated whether or not the variable name is present.

You can access data members in C++ without regard to access restrictions. However, you cannot call member functions.

Writing Functions with Inline Assembly

If you write a function with inline assembly code, it's easy to pass arguments to the function and return a value from it. The following examples compare a function first written for a separate assembler and then rewritten for the inline assembler. The function, called `power2`, receives two parameters, multiplying the first parameter by 2 to the power of the second parameter. Written for a separate assembler, the function might look like this:

```
; POWER.ASM
; Compute the power of an integer
;
       PUBLIC _power2
_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC

        push ebp        ; Save EBP
        mov ebp, esp    ; Move ESP into EBP so we can refer
                        ;   to arguments on the stack
```

```
        mov eax, [ebp+4] ; Get first argument
        mov ecx, [ebp+6] ; Get second argument
        shl eax, cl      ; EAX = EAX * ( 2 ^ CL )
        pop ebp          ; Restore EBP
        ret              ; Return with sum in EAX

_power2 ENDP
_TEXT   ENDS
        END
```

Since it's written for a separate assembler, the function requires a separate source file and assembly and link steps. C and C++ function arguments are usually passed on the stack, so this version of the `power2` function accesses its arguments by their positions on the stack. (Note that the **MODEL** directive, available in MASM and some other assemblers, also allows you to access stack arguments and local stack variables by name.)

The POWER2.C program writes the `power2` function with inline assembly code:

```
/* POWER2.C */
#include <stdio.h>

int power2( int num, int power );

void main( void )
{
   printf( "3 times 2 to the power of 5 is %d\n", \
           power2( 3, 5) );
}
int power2( int num, int power )
{
   __asm
   {
      mov eax, num    ; Get first argument
      mov ecx, power  ; Get second argument
      shl eax, cl     ; EAX = EAX * ( 2 to the power of CL )
   }
   /* Return with result in EAX */
}
```

The inline version of the `power2` function refers to its arguments by name and appears in the same source file as the rest of the program. This version also requires fewer assembly instructions.

Because the inline version of `power2` doesn't execute a C **return** statement, it causes a harmless warning if you compile at warning level 2 or higher. The function does return a value, but the compiler cannot tell that in the absence of a **return** statement. You can use #pragma warning to disable the generation of this warning.

Using and Preserving Registers in Inline Assembly

In general, you should not assume that a register will have a given value when an **__asm** block begins. Register values are not guaranteed to be preserved across separate **__asm** blocks. If you end a block of inline code and begin another, you cannot rely on the registers in the second block to retain their values from the first block. An **__asm** block inherits whatever register values result from the normal flow of control.

If you use the **__fastcall** calling convention, the compiler passes function arguments in registers instead of on the stack. This can create problems in functions with **__asm** blocks because a function has no way to tell which parameter is in which register. If the function happens to receive a parameter in EAX and immediately stores something else in EAX, the original parameter is lost. In addition, you must preserve the ECX register in any function declared with **__fastcall**.

To avoid such register conflicts, don't use the **__fastcall** convention for functions that contain an **__asm** block. If you specify the **__fastcall** convention globally with the /Gr compiler option, declare every function containing an **__asm** block with **__cdecl** or **__stdcall**. (The **__cdecl** attribute tells the compiler to use the C calling convention for that function.) If you are not compiling with /Gr, avoid declaring the function with the **__fastcall** attribute.
When using **__asm** to write assembly language in C/C++ functions, you don't need to preserve the EAX, EBX, ECX, EDX, ESI, or EDI registers. For example, in the POWER2.C example in Writing Functions with Inline Assembly, the `power2` function doesn't preserve the value in the EAX register. However, using these registers will affect code quality because the register allocator cannot use them to store values across **__asm** blocks. In addition, by using EBX, ESI or EDI in inline assembly code, you force the compiler to save and restore those registers in the function prologue and epilogue.
You should preserve other registers you use (such as DS, SS, SP, BP, and flags registers) for the scope of the **__asm** block. You should preserve the ESP and EBP registers unless you have some reason to change them (to switch stacks, for example). Also see Optimizing Inline Assembly.
Note If your inline assembly code changes the direction flag using the STD or CLD instructions, you must restore the flag to its original value.

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.
The following code shows all the permutations:

```
void func( void )
{
   goto C_Dest;  /* Legal: correct case   */
   goto c_dest;  /* Error: incorrect case */

   goto A_Dest;  /* Legal: correct case   */
   goto a_dest;  /* Legal: incorrect case */

   __asm
   {
      jmp C_Dest ; Legal: correct case
      jmp c_dest ; Legal: incorrect case

      jmp A_Dest ; Legal: correct case
      jmp a_dest ; Legal: incorrect case

      a_dest:    ; __asm label
   }

   C_Dest:       /* C label */
   return;
}
```

Don't use C library function names as labels in **__asm** blocks. For instance, you might be tempted to use `exit` as a label, as follows:

```
; BAD TECHNIQUE: using library function name as label
jne exit
   .
   .
   .
exit:
```

```
   ; More __asm code follows
```
Because **exit** is the name of a C library function, this code might cause a jump to the **exit** function instead of to the desired location.
As in MASM programs, the dollar symbol (**$**) serves as the current location counter. It is a label for the instruction currently being assembled. In **__asm** blocks, its main use is to make long conditional jumps:
```
jne $+5 ; next instruction is 5 bytes long
jmp farlabel
; $+5
   .
   .
   .
farlabel:
```

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:
```
#include <stdio.h>

char format[] = "%s %s\n";
char hello[] = "Hello";
char world[] = "world";
void main( void )
{
   __asm
   {
      mov  eax, offset world
      push eax
      mov  eax, offset hello
      push eax
      mov  eax, offset format
      push eax
      call printf
      //clean up the stack so that main can exit cleanly
      //use the unused register ebx to do the cleanup
      pop  ebx
      pop  ebx
      pop  ebx
   }
}
```
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 you call an overloaded global C++ function or a C++ member function, the compiler issues an error.
You 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`).

To illustrate, the following example defines a simple macro:

```
#define PORTIO __asm      \
/* Port output */         \
{                         \
   __asm mov al, 2        \
   __asm mov dx, 0xD007   \
   __asm out al, dx       \
}
```

At first glance, the last three **__asm** keywords seem superfluous. They are needed, however, because the macro expands into a single line:

```
__asm /* Port output */ { __asm mov al, 2  __asm mov dx, 0xD007 __asm out al, dx }
```

The third and fourth **__asm** keywords are needed as statement separators. The only statement separators recognized in **__asm** blocks are the newline character and **__asm** keyword. Because a block defined as a macro is one logical line, you must separate each instruction with **__asm**.

The braces are essential as well. If you omit them, the compiler can be confused by C or C++ statements on the same line to the right of the macro invocation. Without the closing brace, the compiler cannot tell where assembly code stops, and it sees C or C++ statements after the **__asm** block as assembly instructions.

Assembly-style comments that start with a semicolon (**;**) continue to the end of the line. This causes problems in macros because the compiler ignores everything after the comment, all the way to the end of the logical line. The same is true of single-line C or C++ comments (`// comment`). To prevent errors, use old-style C comments (`/* comment */`) in **__asm** blocks defined as macros.

An **__asm** block written as a C macro can take arguments. Unlike an ordinary C macro, however, an **__asm** macro cannot return a value. So you cannot use such macros in C or C++ expressions.

Be careful not to invoke macros of this type indiscriminately. For instance, invoking an assembly-language macro in a function declared with the **__fastcall** convention may cause unexpected results. (See Using and Preserving Registers in Inline Assembly.)

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