Contents

.CODE

COMM

COMMENT

.CONST

.CONTINUE

.CREF

.DATA

.DATA?

DB

DD

DF

.DOSSEG

DOSSEG

DQ

DT

DW

DWORD

ECHO

.ELSE

ELSE

ELSEIF

ELSEIF2

END

.ENDIF

ENDM

ENDP

.ENDPROLOG

ENDS

.ENDW

EQU

.ERR

.ERR2

.ERRB

.ERRDEF

.ERRDIF, .ERRDIFI

.ERRE

.ERRIDN, .ERRIDNI

.ERRNB

.ERRNDEF

.ERRNZ

EVEN

.EXIT

EXITM

EXTERN

EXTERNDEF

EXTRN

.FARDATA

.FARDATA?

FOR

FORC

.FPO

FWORD

GOTO

GROUP

.IF

IF

IF2

IFB

IFDEF

IFDIF, IFDIFI

IFE

IFIDN, IFIDNI

IFNB

IFNDEF

INCLUDE

INCLUDELIB

INSTR

INVOKE

IRP

IRPC

.K3D

LABEL

.LALL

.LFCOND

.LIST

.LISTALL

.LISTIF

.LISTMACRO

.LISTMACROALL

LOCAL

MACRO

MMWORD

.MMX

.MODEL

NAME

.NOCREF

.NOLIST

.NOLISTIF

.NOLISTMACRO

OPTION

ORG

%OUT

OWORD

PAGE

POPCONTEXT

PROC

PROTO

PUBLIC

PURGE

PUSHCONTEXT

.PUSHFRAME

.PUSHREG

QWORD

.RADIX

REAL10

REAL4

REAL8

RECORD

.REPEAT

REPEAT

REPT

.SAFESEH

.SALL

.SAVEREG

.SAVEXMM128

SBYTE

SDWORD

SEGMENT

.SEQ

.SETFRAME

.SFCOND

SIZESTR

SQWORD

.STACK

.STARTUP

STRUC

STRUCT

SUBSTR

Compiler intrinsics

9/21/2020 • 2 minutes to read • Edit Online

Most functions are contained in libraries, but some functions are built in (that is, intrinsic) to the compiler. These are referred to as intrinsic functions or intrinsics.

Remarks

If a function is an intrinsic, the code for that function is usually inserted inline, avoiding the overhead of a function call and allowing highly efficient machine instructions to be emitted for that function. An intrinsic is often faster than the equivalent inline assembly, because the optimizer has a built-in knowledge of how many intrinsics behave, so some optimizations can be available that are not available when inline assembly is used. Also, the optimizer can expand the intrinsic differently, align buffers differently, or make other adjustments depending on the context and arguments of the call.

The use of intrinsics affects the portability of code, because intrinsics that are available in Visual C++ might not be available if the code is compiled with other compilers and some intrinsics that might be available for some target architectures are not available for all architectures. However, intrinsics are usually more portable than inline assembly. The intrinsics are required on 64-bit architectures where inline assembly is not supported.

Some intrinsics, such as `__assume` and `__ReadWriteBarrier`, provide information to the compiler, which affects the behavior of the optimizer.

Some intrinsics are available only as intrinsics, and some are available both in function and intrinsic implementations. You can instruct the compiler to use the intrinsic implementation in one of two ways, depending on whether you want to enable only specific functions or you want to enable all intrinsics. The first way is to use `#pragma intrinsic(` *intrinsic-function-name-list* `)`. The pragma can be used to specify a single intrinsic or multiple intrinsics separated by commas. The second is to use the /Oi (Generate intrinsic functions) compiler option, which makes all intrinsics on a given platform available. Under **/Oi**, use `#pragma function(` *intrinsic-function-name-list* `)` to force a function call to be used instead of an intrinsic. If the documentation for a specific intrinsic notes that the routine is only available as an intrinsic, then the intrinsic implementation is used regardless of whether **/Oi** or `#pragma intrinsic` is specified. In all cases, **/Oi** or `#pragma intrinsic` allows, but does not force, the optimizer to use the intrinsic. The optimizer can still call the function.

Some standard C/C++ library functions are available in intrinsic implementations on some architectures. When calling a CRT function, the intrinsic implementation is used if **/Oi** is specified on the command line.

A header file, <intrin.h>, is available that declares prototypes for the common intrinsic functions. Manufacturer-specific intrinsics are available in the <immintrin.h> and <ammintrin.h> header files. Additionally, certain Windows headers declare functions that map onto a compiler intrinsic.

The following sections list all intrinsics that are available on various architectures. For more information on how the intrinsics work on your particular target processor, refer to the manufacturer's reference documentation.

- ARM intrinsics
- ARM64 intrinsics
- x86 intrinsics list
- x64 (amd64) Intrinsics List
- Intrinsics available on all architectures

- Alphabetical listing of intrinsic functions

See also

ARM assembler reference

Microsoft Macro Assembler reference

Keywords

C run-time library reference

Inline Assembler

3/27/2020 • 2 minutes to read • Edit Online

Microsoft Specific

Assembly language serves many purposes, such as improving program speed, reducing memory needs, and controlling hardware. You can use the inline assembler to embed assembly-language instructions directly in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler, so you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

NOTE

Programs with inline assembler code are not fully portable to other hardware platforms. If you are designing for portability, avoid using inline assembler.

Inline assembly is not supported on the ARM and x64 processors. The following topics explain how to use the Visual C/C++ inline assembler with x86 processors:

- Inline Assembler Overview
- Advantages of Inline Assembly
- __asm
- Using Assembly Language in __asm Blocks
- Using C or C++ in __asm Blocks
- Using and Preserving Registers in Inline Assembly
- Jumping to Labels in Inline Assembly
- Calling C Functions in Inline Assembly
- Calling C++ Functions in Inline Assembly
- Defining __asm Blocks as C Macros
- Optimizing Inline Assembly

END Microsoft Specific

See also

Compiler Intrinsics and Assembly Language
C++ Language Reference

Inline Assembler Overview

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

The inline assembler lets you embed assembly-language instructions in your C and C++ source programs without extra assembly and link steps. The inline assembler is built into the compiler — you don't need a separate assembler such as the Microsoft Macro Assembler (MASM).

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 or C++ variable or function name that is in scope, so it is easy to integrate it with your program's C and C++ code. And because the assembly code can be mixed with C and C++ statements, it can do tasks that are cumbersome or impossible in C or C++ alone.

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 is a simple `__asm` block enclosed in braces. (The code is a custom function prolog sequence.)

```
// asm_overview.cpp
// processor: x86
void __declspec(naked) main()
{
    // Naked functions must provide their own prolog...
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
    }

    // ... and epilog
    __asm {
        pop ebp
        ret
    }
}
```

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

```
__asm push ebp
__asm mov  ebp, esp
__asm sub  esp, __LOCAL_SIZE
```

Since the `__asm` keyword is a statement separator, you can also put assembly instructions on the same line:

```
__asm push ebp   __asm mov  ebp, esp   __asm sub  esp, __LOCAL_SIZE
```

END Microsoft Specific

See also

Advantages of Inline Assembly

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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-optimizing 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.

END Microsoft Specific

See also

Inline Assembler

__asm

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

> **NOTE**
> Visual C++ support for the Standard C++ `asm` keyword is limited to the fact that the compiler will not generate an error on the keyword. However, an `asm` block will not generate any meaningful code. Use `__asm` instead of `asm`.

Grammar

asm-block:
  `__asm` *assembly-instruction* `;`$_{opt}$
  `__asm {` *assembly-instruction-list* `}` `;`$_{opt}$

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

Remarks

If used without braces, the `__asm` keyword means that the rest of the line is an assembly-language statement. If used with braces, it means that each line between the braces is an assembly-language statement. For compatibility with previous versions, `_asm` is a synonym for `__asm`.

Since the `__asm` keyword is a statement separator, you can put assembly instructions on the same line.

Before Visual Studio 2005, the instruction

```
__asm int 3
```

did not cause native code to be generated when compiled with **/clr**; the compiler translated the instruction to a CLR break instruction.

`__asm int 3` now results in native code generation for the function. If you want a function to cause a break point in your code and if you want that function compiled to MSIL, use __debugbreak.

For compatibility with previous versions, `_asm` is a synonym for `__asm` unless compiler option /Za (Disable language extensions) is specified.

Example

The following code fragment is a simple `__asm` block enclosed in braces:

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

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
```

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.

END Microsoft Specific

See also

Keywords
Inline Assembler

Using Assembly Language in __asm Blocks

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

The inline assembler has much in common with other assemblers. For example, it accepts any expression that is legal in MASM. This section describes the use of assembly-language features in `__asm` blocks.

What do you want to know more about?

- Instruction Set for Inline Assembly
- MASM Expressions in Inline Assembly
- Data Directives and Operators in Inline Assembly
- EVEN and ALIGN Directives
- MASM Macro Directives in Inline Assembly
- Segment References in Inline Assembly
- Type and Variable Sizes in Inline Assembly
- Assembly-Language Comments
- The _emit Pseudoinstruction
- Debugging and Listings for Inline Assembly
- Intel's MMX Instruction Set

END Microsoft Specific

See also

Inline Assembler

Instruction Set for Inline Assembly

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

The Microsoft C++ compiler supports all opcodes through the Pentium 4 and AMD Athlon. Additional instructions supported by the target processor can be created with the _emit Pseudoinstruction.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

MASM Expressions in Inline Assembly

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Data Directives and Operators in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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**.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

EVEN and ALIGN Directives

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

MASM Macro Directives in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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. See Using C or C++ in __asm Blocks for more information.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Segment References in Inline Assembly

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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].

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Type and Variable Sizes in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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)`	32
TYPE arr	`sizeof(arr[0])`	4

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Assembly-Language Comments

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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. (See Defining __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.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

_emit Pseudoinstruction

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

The **_emit** pseudoinstruction defines one byte at the current location in the current text segment. The **_emit** pseudoinstruction resembles the DB directive of MASM.

The following fragment places the bytes 0x4A, 0x43, and 0x4B into the code:

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

Caution

If `_emit` generates instructions that modify registers, and you compile the application with optimizations, the compiler cannot determine what registers are affected. For example, if `_emit` generates an instruction that modifies the **rax** register, the compiler does not know that **rax** has changed. The compiler might then make an incorrect assumption about the value in that register after the inline assembler code executes. Consequently, your application might exhibit unpredictable behavior when it runs.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Debugging and Listings for Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Intel's MMX Instruction Set

3/24/2020 • 2 minutes to read • Edit Online

Microsoft Specific

The Microsoft 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 compiler generates a warning message if the function contains MMX instructions but does not contain an EMMS instruction to empty the multimedia state. For more information, see the Intel Web site.

END Microsoft Specific

See also

Using Assembly Language in __asm Blocks

Using C or C++ in __asm Blocks

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

What do you want to know more about?

- Using Operators in __asm Blocks
- Using C or C++ Symbols_in __asm Blocks
- Accessing C or C++ Data in __asm Blocks
- Writing Functions with Inline Assembly

END Microsoft Specific

See also

Inline Assembler

Using Operators in __asm Blocks

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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+24 (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 + 24

array[6] = 0;                   /* Store 0 at array + 24 */
```

END Microsoft Specific

See also

Using C or C++ in __asm Blocks

Using C or C++ Symbols in __asm Blocks

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

END Microsoft Specific

See also

Using C or C++ in __asm Blocks

Accessing C or C++ Data in __asm Blocks

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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 structure types in the following sample share `same_name` as their member 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:

```
// InlineAssembler_Accessing_C_asm_Blocks.cpp
// processor: x86
#include <stdio.h>
struct first_type
{
   char *weasel;
   int same_name;
};

struct second_type
{
   int wonton;
   long same_name;
};

int main()
{
   struct first_type hal;
   struct second_type oat;

   __asm
   {
      lea ebx, hal
      mov ecx, [ebx]hal.same_name ; Must use 'hal'
      mov esi, [ebx].weasel       ; Can omit 'hal'
   }
   return 0;
}
```

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.

END Microsoft Specific

See also

Using C or C++ in __asm Blocks

Writing Functions with Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.)

Example

This program writes the `power2` function with inline assembly code:

```
// Power2_inline_asm.c
// compile with: /EHsc
// processor: x86

#include <stdio.h>

int power2( int num, int power );

int main( void )
{
    printf_s( "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.

END Microsoft Specific

See also

Using C or C++ in __asm Blocks

Using and Preserving Registers in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

Some SSE types require eight-byte stack alignment, forcing the compiler to emit dynamic stack-alignment code. To be able to access both the local variables and the function parameters after the alignment, the compiler maintains two frame pointers. If the compiler performs frame pointer omission (FPO), it will use EBP and ESP. If the compiler does not perform FPO, it will use EBX and EBP. To ensure code runs correctly, do not modify EBX in asm code if the function requires dynamic stack alignment as it could modify the frame pointer. Either move the eight-byte aligned types out of the function, or avoid using EBX.

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.

END Microsoft Specific

See also

Inline Assembler

Jumping to Labels in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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;
}
int main()
{
}
```

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:
```

END Microsoft Specific

See also

Inline Assembler

Calling C Functions in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

An `__asm` block can call C functions, including C library routines. The following example calls the `printf` library routine:

```
// InlineAssembler_Calling_C_Functions_in_Inline_Assembly.cpp
// processor: x86
#include <stdio.h>

char format[] = "%s %s\n";
char hello[] = "Hello";
char world[] = "world";
int 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`.

END Microsoft Specific

See also

Inline Assembler

Calling C++ Functions in Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

END Microsoft Specific

See also

Inline Assembler

Defining __asm Blocks as C Macros

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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 dx, al       \
}
```

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 dx, al }
```

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.)

END Microsoft Specific

See also

Inline Assembler

Optimizing Inline Assembly

9/21/2020 • 2 minutes to read • Edit Online

Microsoft Specific

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.

END Microsoft Specific

See also

Inline Assembler

ARM Assembler Reference

11/18/2019 • 2 minutes to read • Edit Online

The articles in this section of the documentation provide reference material for the Microsoft ARM assembler (armasm) and related tools.

Related Articles

TITLE	DESCRIPTION
ARM Assembler Command-Line Reference	Describes the armasm command-line options.
ARM Assembler Diagnostic Messages	Describes commonly encountered armasm warning and error messages.
ARM Assembler Directives	Describes the ARM directives that are different in armasm.
ARM Architecture Reference Manual on the ARM Developer website.	Choose the relevant manual for your ARM architecture. Each contains reference sections about ARM, Thumb, NEON, and VFP, and additional information about the ARM assembly language.
ARM Compiler armasm User Guide on the ARM Developer website.	Choose a recent version to find up-to-date information about the ARM assembly language. **Note:** The "armasm" assembler that is referred to on the ARM Developer website is not the Microsoft armasm assembler that is included in Visual Studio and is documented in this section.

See also

ARM intrinsics
ARM64 intrinsics
Compiler intrinsics

ARM Assembler command-line reference

2/12/2020 • 2 minutes to read • Edit Online

This article provides command-line information about the Microsoft ARM assembler, **armasm**. **armasm** assembles ARMv7 Thumb assembly language into the Microsoft implementation of the Common Object File Format (COFF). The linker can link COFF code objects produced by both the ARM assembler and the C compiler. It can link either together with object libraries created by the librarian.

Syntax

> `armasm` [*options*] *source_file object_file*
> `armasm` [*options*] `-o` *object_file source_file*

Parameters

options
A combination of zero or more of the following options:

- `-errors` *filename*
  Redirect error and warning messages to *filename*.

- `-i` *dir*[`;` *dir*]
  Add the specified directories to the include search path.

- `-predefine` *directive*
  Specify a SETA, SETL, or SETS directive to predefine a symbol.
  Example: `armasm.exe -predefine "COUNT SETA 150" source.asm`
  For more information, see the ARM Compiler armasm Reference Guide.

- `-nowarn`
  Disable all warning messages.

- `-ignore` *warning*
  Disable the specified warning. For possible values, see the section about warnings.

- `-help`
  Print the command-line help message.

- `-machine` *machine*
  Specify the machine type to set in the PE header. Possible values for *machine* are:
  ARM—Sets the machine type to IMAGE_FILE_MACHINE_ARMNT. This option is the default.
  THUMB—Sets the machine type to IMAGE_FILE_MACHINE_THUMB.

- `-oldit`
  Generate ARMv7-style IT blocks. By default, ARMv8-compatible IT blocks are generated.

- `-via` *filename*
  Read additional command-line arguments from *filename*.

- `-16`
  Assemble source as 16-bit Thumb instructions. This option is the default.

- `-32`

Assemble source as 32-bit ARM instructions.

- `-g`
  Generate debugging information.

- `-errorReport:` *option*
  This option is deprecated. Starting in Windows Vista, error reporting is controlled by Windows Error Reporting (WER) settings.

source_file
The name of the source file.

object_file
The name of the object (output) file.

Remarks

The following example demonstrates how to use armasm in a typical scenario. First, use armasm to build an assembly language source (.asm) file to an object (.obj) file. Then, use the CL command-line C compiler to compile a source (.c) file, and also specify the linker option to link the ARM object file.

```
armasm myasmcode.asm -o myasmcode.obj
cl myccode.c /link myasmcode.obj
```

See also

ARM Assembler diagnostic messages
ARM Assembler directives

ARM Assembler diagnostic messages

5/31/2019 • 3 minutes to read • Edit Online

The Microsoft ARM assembler (*armasm*) emits diagnostic warnings and errors when it encounters them. This article describes the most commonly-encountered messages.

Syntax

> *filename*(*line-number*) : [**error**|**warning**] **A***number*: *message*

Diagnostic messages - Errors

> A2193: this instruction generates unpredictable behavior

The ARM architecture cannot guarantee what happens when this instruction is executed. For details about the well-defined forms of this instruction, consult the ARM Architecture Reference Manual.

```
    ADD r0, r8, pc         ; A2193: this instruction generates unpredictable behavior
```

> A2196: instruction cannot be encoded in 16 bits

The specified instruction cannot be encoded as a 16-bit Thumb instruction. Specify a 32-bit instruction, or rearrange code to bring the target label into the range of a 16-bit instruction.

The assembler may attempt to encode a branch in 16 bits and fail with this error, even though a 32-bit branch is encodable. You can solve this problem by using the `.W` specifier to explicitly mark the branch as 32-bit.

```
    ADD.N r0, r1, r2      ; A2196: instruction cannot be encoded in 16 bits

    B.W label             ; OK
    B.N label             ; A2196: instruction cannot be encoded in 16 bits
    SPACE 10000
label
```

> A2202: Pre-UAL instruction syntax not allowed in THUMB region

Thumb code must use the Unified Assembler Language (UAL) syntax. The old syntax is no longer accepted

```
    ADDEQS r0, r1         ; A2202: Pre-UAL instruction syntax not allowed in THUMB region
    ADDSEQ r0, r1         ; OK
```

> A2513: Rotation must be even

In ARM mode, there is an alternate syntax for specifying constants. Instead of writing `#<const>`, you can write `#<byte>,#<rot>`, which represents the constant value that is obtained by rotating the value `<byte>` right by `<rot>`. When you use this syntax, you must make the value of `<rot>` even.

```
    MOV r0, #4, #2       ; OK
    MOV r0, #4, #1       ; A2513: Rotation must be even
```

A2557: Incorrect number of bytes to write back

On the NEON structure load and store instructions (`VLDn`, `VSTn`), there is an alternate syntax for specifying writeback to the base register. Instead of putting an exclamation point (!) after the address, you can specify an immediate value that indicates the offset to be added to the base register. If you use this syntax, you must specify the exact number of bytes that were loaded or stored by the instruction.

```
    VLD1.8 {d0-d3}, [r0]!         ; OK
    VLD1.8 {d0-d3}, [r0], #32     ; OK
    VLD1.8 {d0-d3}, [r0], #100    ; A2557: Incorrect number of bytes to write back
```

Diagnostic messages - Warnings

A4228: Alignment value exceeds AREA alignment; alignment not guaranteed

The alignment that is specified in an `ALIGN` directive is greater than the alignment of the enclosing `AREA`. As a result, the assembler cannot guarantee that the `ALIGN` directive will be honored.

To fix this, you can specify on the `AREA` directive an `ALIGN` attribute that is equal to or greater than the desired alignment.

```
AREA |.myarea1|
ALIGN 8           ; A4228: Alignment value exceeds AREA alignment; alignment not guaranteed

AREA |.myarea2|,ALIGN=3
ALIGN 8           ; OK
```

A4508: Use of this rotated constant is deprecated

In ARM mode, there is an alternate syntax for specifying constants. Instead of writing `#<const>`, you can write `#<byte>,#<rot>`, which represents the constant value that is obtained by rotating the value `<byte>` right by `<rot>`. In some contexts, ARM has deprecated the use of these rotated constants. In these cases, use the basic `#<const>` syntax instead.

```
    ANDS r0, r0, #1               ; OK
    ANDS r0, r0, #4, #2           ; A4508: Use of this rotated constant is deprecated
```

A4509: This form of conditional instruction is deprecated

This form of conditional instruction has been deprecated by ARM in the ARMv8 architecture. We recommend that you change the code to use conditional branches. To see which conditional instructions are still supported, consult the ARM Architecture Reference Manual.

This warning is not emitted when the **-oldit** command-line switch is used.

```
    ADDEQ r0, r1, r8             ; A4509: This form of conditional instruction is deprecated
```

See also

ARM Assembler Command-Line Reference

ARM Assembler Directives

ARM Assembler Directives

10/31/2018 • 2 minutes to read • Edit Online

For the most part, the Microsoft ARM assembler uses the ARM assembly language, which is documented in the ARM Compiler armasm Reference Guide. However, the Microsoft implementations of some assembly directives differ from the ARM assembly directives. This article explains the differences.

Microsoft Implementations of ARM Assembly Directives

- AREA

  The Microsoft ARM assembler supports these `AREA` attributes: `ALIGN`, `CODE`, `CODEALIGN`, `DATA`, `NOINIT`, `READONLY`, `READWRITE`, `THUMB`, `ARM`.

  All except `THUMB` and `ARM` work as documented in the ARM Compiler armasm Reference Guide.

  In the Microsoft ARM assembler, `THUMB` indicates that a `CODE` section contains Thumb code, and is the default for `CODE` sections. `ARM` indicates that the section contains ARM code.

- ATTR

  Not supported.

- CODE16

  Not supported because it implies pre-UAL Thumb syntax, which the Microsoft ARM assembler does not allow. Use the `THUMB` directive instead, along with UAL syntax.

- COMMON

  Specification of an alignment for the common region is not supported.

- DCDO

  Not supported.

- `DN`, `QN`, `SN`

  Specification of a type or a lane on the register alias is not supported.

- ENTRY

  Not supported.

- EQU

  Specification of a type for the defined symbol is not supported.

- `EXPORT` and `GLOBAL`

  Specifies exports using this syntax:

  > EXPORT|GLOBAL *sym*{[*type*]}

  sym is the symbol to be exported. [*type*], if specified, can be either `[DATA]` to indicate that the symbol points to data or `[FUNC]` to indicate that the symbol points to code. `GLOBAL` is a synonym for `EXPORT`.

- EXPORTAS

  Not supported.

- FRAME

  Not supported.

- `FUNCTION` and `PROC`

  Although the assembly syntax supports the specification of a custom calling convention on procedures by listing the registers that are caller-save and those that are callee-save, the Microsoft ARM assembler accepts the syntax but ignores the register lists. The debug information that is produced by the assembler supports only the default calling convention.

- `IMPORT` and `EXTERN`

  Specifies imports using this syntax:

  > **IMPORT|EXTERN** *sym*{**,** **WEAK** *alias*{**,** **TYPE** *t*}}

  sym is the name of the symbol to be imported.

  If `WEAK` *alias* is specified, it indicates that *sym* is a weak external. If no definition for it is found at link time, then all references to it bind instead to *alias*.

  If `TYPE` *t* is specified, then *t* indicates how the linker should attempt to resolve *sym*. These values for *t* are possible:

VALUE	DESCRIPTION
1	Do not perform a library search for *sym*
2	Perform a library search for *sym*
3	*sym* is an alias for *alias* (default)

  `EXTERN` is a synonym for `IMPORT`, except that *sym* is imported only if there are references to it in the current assembly.

- MACRO

  The use of a variable to hold the condition code of a macro is not supported. Default values for macro parameters are not supported.

- NOFP

  Not supported.

- `OPT`, `TTL`, `SUBT`

  Not supported because the Microsoft ARM assembler does not produce listings.

- PRESERVE8

  Not supported.

- RELOC

  `RELOC n` can only follow an instruction or a data definition directive. There is no "anonymous symbol" that can be relocated.

- REQUIRE

  Not supported.

- REQUIRE8

  Not supported.

- THUMBX

  Not supported because the Microsoft ARM assembler does not support the Thumb-2EE instruction set.

See also

ARM Assembler Command-Line Reference
ARM Assembler Diagnostic Messages

Microsoft Macro Assembler reference

12/20/2019 • 2 minutes to read • Edit Online

The Microsoft Macro Assembler (MASM) provides several advantages over inline assembly. MASM contains a macro language that has features such as looping, arithmetic, and text string processing. MASM also gives you greater control over the hardware because it supports the instruction sets of the 386, 486, and Pentium processors. By using MASM, you also can reduce time and memory overhead.

In This Section

ML and ML64 command-line option
Describes the ML.exe and ML64.exe command-line options.

ML error messages
Describes ML.exe fatal and nonfatal error messages and warnings.

Directives reference
Provides links to articles that discuss the use of directives in MASM.

Symbols Reference
Provides links to articles that discuss the use of symbols in MASM.

Operators Reference
Provides links to articles that discuss the use of operators in MASM.

Processor Manufacturer Programming Manuals
Provides links to websites that may contain programming information about processors that are not manufactured, sold, or supported by Microsoft.

MASM for x64 (ml64.exe)
Information about how to create output files for x64.

MASM BNF Grammar

Formal BNF description of MASM for x64.

Related Sections

C++ in Visual Studio
Provides links to different areas of the Visual Studio and Visual C++ documentation.

See also

Compiler Intrinsics
x86Intrinsics
x64 (amd64) Intrinsics

MASM for x64 (ml64.exe)

12/20/2019 • 3 minutes to read • Edit Online

Visual Studio includes both 32-bit and 64-bit hosted versions of Microsoft Assembler (MASM) to target x64 code. Named ml64.exe, this is the assembler that accepts x64 assembler language. The MASM command-line tools are installed when you choose a C++ workload during Visual Studio installation. The MASM tools are not available as a separate download. For instructions on how to download and install a copy of Visual Studio, see Install Visual Studio. If you do not want to install the complete Visual Studio IDE, but only want the command-line tools, download the Build Tools for Visual Studio.

To use MASM to build code for x64 targets on the command line, you must use a developer command prompt for x64 targets, which sets the required path and other environment variables. For information on how to start a developer command prompt, see Build C/C++ code on the command line.

For information on ml64.exe command line options, see ML and ML64 Command-Line Reference.

Inline assembler or use of the ASM keyword is not supported for x64 or ARM targets. To port your x86 code that uses inline assembler to x64 or ARM, you can convert your code to C++, use compiler intrinsics, or create assembler-language source files. The Microsoft C++ compiler supports intrinsics to allow you to use special-function instructions, for example, privileged, bit scan/test, interlocked, and so on, in as close to a cross-platform manner as possible. For information on available intrinsics, see Compiler Intrinsics.

Add an assembler-language file to a Visual Studio C++ project

The Visual Studio project system supports assembler-language files built by using MASM in your C++ projects. You can create x64 assembler-language source files and build them into object files by using MASM, which supports x64 fully. You can then link these object files to your C++ code built for x64 targets. This is one way to overcome the lack of an x64 inline assembler.

To add an assembler-language file to an existing Visual Studio C++ project

1. Select the project in **Solution Explorer**. On the menu bar, choose **Project**, **Build Customizations**.
2. In the **Visual C++ Build Customization Files** dialog box, check the checkbox next to **masm(.targets,.props)**. Choose **OK** to save your selection and close the dialog box.
3. On the menu bar, choose **Project**, **Add New Item**.
4. In the **Add New Item** dialog box, select **C++ file (.cpp)** in the center pane. In the **Name** edit control, enter a new file name that has a **.asm** extension instead of .cpp. Choose **Add** to add the file to your project and close the dialog box.

Create your assembler-language code in the .asm file you added. When you build your solution, the MASM assembler is invoked to assemble the .asm file into an object file that is then linked into your project. To make symbol access easier, declare your assembler functions as `extern "C"` in your C++ source code, rather than using the C++ name decoration conventions in your assembler-language source files.

ml64-Specific Directives

You can use the following ml64-specific directives in your assembler-language source code that targets x64:

- .ALLOCSTACK
- .ENDPROLOG

- .PUSHFRAME
- .PUSHREG
- .SAVEREG
- .SAVEXMM128
- .SETFRAME

In addition, the PROC directive has been updated for use with ml64.exe.

32-Bit Address Mode (Address Size Override)

MASM emits the 0x67 address size override if a memory operand includes 32-bit registers. For example, the following examples cause the address size override to be emitted:

```
mov rax, QWORD PTR [ecx]
mov eax, DWORD PTR [ecx*2+r10d]
mov eax, DWORD PTR [ecx*2+r10d+0100h]
prefetch [eax]
movnti rax, QWORD PTR [r8d]
```

MASM assumes that if a 32-bit displacement appears alone as a memory operand, 64-bit addressing is intended. There is currently no support for 32-bit addressing with such operands.

Finally, mixing register sizes within a memory operand, as demonstrated in the following code, generates an error.

```
mov eax, DWORD PTR [rcx*2+r10d]
mov eax, DWORD PTR [ecx*2+r10+0100h]
```

See also

Microsoft Macro Assembler Reference

ML and ML64 command-line reference

2/12/2020 • 3 minutes to read • Edit Online

Assembles and links one or more assembly-language source files. The command-line options are case-sensitive.

For more information on ml64.exe, see MASM for x64 (ml64.exe).

Syntax

> ML [*options*] *filename* [[*options*] *filename*]
>
> ML64 [*options*] *filename* [[*options*] *filename*] ... [/link *link_options*]

Parameters

options
The options listed in the following table.

OPTION	ACTION
/AT	Enables tiny-memory-model support. Enables error messages for code constructs that violate the requirements for .com format files. This option isn't equivalent to the .MODEL **TINY** directive. Not available in ml64.exe.
/Bl *filename*	Selects an alternate linker.
/c	Assembles only. Does no linking.
/coff	Generates common object file format (COFF) type of object module. Required for Win32 assembly language development. Not available in ml64.exe.
/Cp	Preserves case of all user identifiers.
/Cu	Maps all identifiers to upper case (default). Not available in ml64.exe.
/Cx	Preserves case in public and extern symbols.
/D *symbol*□=*value*□	Defines a text macro with the given name. If *value* is missing, it's blank. Multiple tokens separated by spaces must be enclosed in quotation marks.
/EP	Generates a preprocessed source listing (sent to STDOUT). See **/Sf**.
/ERRORREPORT [**NONE** | **PROMPT** | **QUEUE** | **SEND**]	Deprecated. Error reporting is controlled by Windows Error Reporting (WER) settings.

OPTION	ACTION
/F *hexnum*	Sets stack size to *hexnum* bytes (the same as **/link /STACK**:*number*). The value must be expressed in hexadecimal notation. There must be a space between **/F** and *hexnum*.
/Fe *filename*	Names the executable file.
/Fl⟦*filename*⟧	Generates an assembled code listing. See **/Sf**.
/Fm⟦*filename*⟧	Creates a linker map file.
/Fo *filename*	Names an object file. For more information, see Remarks.
/FPi	Generates emulator fix-ups for floating-point arithmetic (mixed language only). Not available in ml64.exe.
/Fr⟦*filename*⟧	Generates a source browser .sbr file.
/FR⟦*filename*⟧	Generates an extended form of a source browser .sbr file.
/Gc	Specifies use of FORTRAN- or Pascal-style function calling and naming conventions. Same as **OPTION LANGUAGE:PASCAL**. Not available in ml64.exe.
/Gd	Specifies use of C-style function calling and naming conventions. Same as **OPTION LANGUAGE:C**. Not available in ml64.exe.
/GZ	Specifies use of __stdcall function calling and naming conventions. Same as **OPTION LANGUAGE:STCALL**. Not available in ml64.exe.
/H *number*	Restricts external names to number significant characters. The default is 31 characters. Not available in ml64.exe.
/help	Calls QuickHelp for help on ML.
/I *pathname*	Sets path for include file. A maximum of 10 **/I** options is allowed.
/nologo	Suppresses messages for successful assembly.
/omf	Generates object module file format (OMF) type of object module. **/omf** implies **/c**; ML.exe doesn't support linking OMF objects. Not available in ml64.exe.

OPTION	ACTION
/Sa	Turns on listing of all available information.
/safeseh	Marks the object as either containing no exception handlers or containing exception handlers that are all declared with .SAFESEH. Not available in ml64.exe.
/Sf	Adds first-pass listing to listing file.
/Sl *width*	Sets the line width of source listing in characters per line. Range is 60 to 255 or 0. Default is 0. Same as PAGE width.
/Sn	Turns off symbol table when producing a listing.
/Sp *length*	Sets the page length of source listing in lines per page. Range is 10 to 255 or 0. Default is 0. Same as PAGE length.
/Ss *text*	Specifies text for source listing. Same as SUBTITLE text.
/St *text*	Specifies title for source listing. Same as TITLE text.
/Sx	Turns on false conditionals in listing.
/Ta *filename*	Assembles source file whose name doesn't end with the .asm extension.
/w	Same as **/W0/WX**.
/W *level*	Sets the warning level, where *level* = 0, 1, 2, or 3.
/WX	Returns an error code if warnings are generated.
/X	Ignore INCLUDE environment path.
/Zd	Generates line-number information in object file.
/Zf	Makes all symbols public.
/Zi	Generates CodeView information in object file.
/Zm	Enables**M510** option for maximum compatibility with MASM 5.1. Not available in ml64.exe.
/Zp□*alignment*□	Packs structures on the specified byte boundary. The *alignment* can be 1, 2, or 4.
/Zs	Performs a syntax check only.
/?	Displays a summary of ML command-line syntax.

filename
The name of the file.

link_options
The link options. For more information, see Linker options.

Remarks

Some command-line options to ML and ML64 are placement-sensitive. For example, because ML and ML64 can accept several **/c** options, any corresponding **/Fo** options must be specified before **/c**. The following command-line example illustrates an object file specification for each assembly file specification:

```
ml.exe /Fo a1.obj /c a.asm /Fo b1.obj /c b.asm
```

Environment Variables

VARIABLE	DESCRIPTION
INCLUDE	Specifies search path for include files.
ML	Specifies default command-line options.
TMP	Specifies path for temporary files.

See also

ML Error Messages
Microsoft Macro Assembler Reference

Directives Reference

9/21/2020 • 2 minutes to read • Edit Online

x64

`.ALLOCSTACK`
`.ENDPROLOG`
`PROC`

`.PUSHFRAME`
`.PUSHREG`
`.SAVEREG`

`.SAVEXMM128`
`.SETFRAME`

Code Labels

`ALIGN`
`EVEN`

`LABEL`

`ORG`

Conditional Assembly

`ELSE`
`ELSEIF`
`ELSEIF2`
`IF`
`IF2`

`IFB`
`IFNB`
`IFDEF`
`IFNDEF`
`IFDIF`

`IFDIFI`
`IFE`
`IFIDN`
`IFIDNI`

Conditional Control Flow

`.BREAK`
`.CONTINUE`
`.ELSE`
`.ELSEIF`

`.ENDIF`
`.ENDW`
`.IF`
`.REPEAT`

`.UNTIL`
`.UNTILCXZ`
`.WHILE`

Conditional Error

`.ERR`
`.ERR2`
`.ERRB`
`.ERRDEF`

`.ERRDIF`
`.ERRDIFI`
`.ERRE`
`.ERRIDN`

`.ERRIDNI`
`.ERRNB`
`.ERRNDEF`
`.ERRNZ`

Data Allocation

`ALIGN`
`BYTE`
`SBYTE`
`DWORD`
`SDWORD`
`EVEN`

`FWORD`
`LABEL`
`ORG`
`QWORD`
`REAL4`

`REAL8`
`REAL10`
`TBYTE`
`WORD`
`SWORD`

Equates

`=`

`EQU`

`TEXTEQU`

Listing Control

`.CREF`
`.LIST`
`.LISTALL`
`.LISTIF`
`.LISTMACRO`

`.LISTMACROALL`
`.NOCREF`
`.NOLIST`
`.NOLISTIF`
`.NOLISTMACRO`

`PAGE`
`SUBTITLE`
`.TFCOND`
`TITLE`

Macros

`ENDM`
`EXITM`

`GOTO`
`LOCAL`

`MACRO`
`PURGE`

Miscellaneous

`ALIAS`
`ASSUME`
`COMMENT`
`ECHO`
`END`
`.FPO`

`INCLUDE`
`INCLUDELIB`
`MMWORD`
`OPTION`
`POPCONTEXT`

`PUSHCONTEXT`
`.RADIX`
`.SAFESEH`
`XMMWORD`
`YMMWORD`

Procedures

`ENDP`

`INVOKE`

`PROC`

`PROTO`

Processor

`.386`
`.386P`
`.387`
`.486`

`.486P`
`.586`
`.586P`
`.686`

`.686P`
`.K3D`
`.MMX`
`.XMM`

Repeat Blocks

`ENDM`
`FOR`

`FORC`
`GOTO`

`REPEAT`
`WHILE`

Scope

`COMM`
`EXTERN`

`EXTERNDEF`
`INCLUDELIB`

`PUBLIC`

Segment

`.ALPHA`
`ASSUME`
`.DOSSEG`

`END`
`ENDS`
`GROUP`

`SEGMENT`
`.SEQ`

Simplified Segment

`.CODE`
`.CONST`
`.DATA`
`.DATA?`

`.DOSSEG`
`.EXIT`
`.FARDATA`
`.FARDATA?`

`.MODEL`
`.STACK`
`.STARTUP`

String

`CATSTR`
`INSTR`

`SIZESTR`

`SUBSTR`

Structure and Record

`ENDS`
`RECORD`

`STRUCT`
`TYPEDEF`

`UNION`

See also

Microsoft Macro Assembler reference
MASM BNF Grammar

=

12/20/2019 • 2 minutes to read • Edit Online

Assigns the numeric value of *expression* to *name*.

Syntax

> *name* = *expression*

Remarks

The symbol can be redefined later.

See also

Directives reference
MASM BNF Grammar

.386 (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of nonprivileged instructions for the 80386 processor; disables assembly of instructions introduced with later processors. (32-bit MASM only.)

Syntax

```
.386
```

Remarks

Also enables 80387 instructions.

See also

Directives Reference
MASM BNF Grammar

.386P (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of all instructions (including privileged) for the 80386 processor; disables assembly of instructions introduced with later processors. (32-bit MASM only.)

Syntax

```
.386P
```

Remarks

Also enables 80387 instructions.

See also

Directives Reference
MASM BNF Grammar

.387 (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of instructions for the 80387 coprocessor. (32-bit MASM only.)

Syntax

```
.387
```

See also

Directives Reference
MASM BNF Grammar

.486 (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of nonprivileged instructions for the 80486 processor. (32-bit MASM only.)

Syntax

```
.486
```

See also

Directives Reference
MASM BNF Grammar

.486P (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of all instructions (including privileged) for the 80486 processor. (32-bit MASM only.)

Syntax

```
.486P
```

See also

Directives Reference
MASM BNF Grammar

.586 (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of nonprivileged instructions for the Pentium processor. (32-bit MASM only.)

Syntax

```
.586
```

See also

Directives Reference
MASM BNF Grammar

.586P (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of all instructions (including privileged) for the Pentium processor. (32-bit MASM only.)

Syntax

```
.586P
```

See also

Directives Reference
MASM BNF Grammar

.686 (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of nonprivileged instructions for the Pentium Pro processor. (32-bit MASM only.)

Syntax

```
.686
```

See also

Directives Reference
MASM BNF Grammar

.686P (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of all instructions (including privileged) for the Pentium Pro processor. (32-bit MASM only.)

Syntax

```
.686P
```

See also

Directives Reference
MASM BNF Grammar

ALIAS

12/20/2019 • 2 minutes to read • Edit Online

The **ALIAS** directive creates an alternate name for a function. This lets you create multiple names for a function, or create libraries that allow the linker (LINK.exe) to map an old function to a new function.

Syntax

```
ALIAS <alias> = <actual-name>
```

Parameters

actual-name
The actual name of the function or procedure. The angle brackets are required.

alias
The alternate or alias name. The angle brackets are required.

See also

Directives Reference
MASM BNF Grammar

ALIGN

12/20/2019 • 2 minutes to read • Edit Online

The **ALIGN** directive aligns the next data element or instruction on an address that is a multiple of its parameter. The parameter must be a power of 2 (for example, 1, 2, 4, and so on) that is less than or equal to the segment alignment.

Syntax

> **ALIGN** ⟦*constantExpression*⟧

Remarks

The **ALIGN** directive allows you to specify the beginning offset of a data element or an instruction. Aligned data can improve performance, at the expense of wasted space between data elements. Large performance improvements can be seen when data accesses are on boundaries that fit within cache lines. Accesses on natural boundaries for native types means less time spent in internal hardware realignment microcode.

The need for aligned instructions is rare on modern processors that use a flat addressing model, but may be required for jump targets in older code for other addressing models.

When data is aligned, the skipped space is padded with zeroes. When instructions are aligned, the skipped space is filled with appropriately-sized NOP instructions.

See also

EVEN
Directives reference
MASM BNF Grammar

.ALLOCSTACK

12/20/2019 • 2 minutes to read • Edit Online

Generates a **UWOP_ALLOC_SMALL** or a **UWOP_ALLOC_LARGE** with the specified size for the current offset in the prologue.

Syntax

> .ALLOCSTACK *size*

Remarks

MASM will choose the most efficient encoding for a given size.

.ALLOCSTACK allows ml64.exe users to specify how a frame function unwinds and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. **.ALLOCSTACK** should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

The *size* operand must be a multiple of 8.

For more information, see MASM for x64 (ml64.exe).

Sample

The following sample shows how to specify an unwind/exception handler:

```
; ml64 ex3.asm /link /entry:Example1  /SUBSYSTEM:Console
text SEGMENT
PUBLIC Example3
PUBLIC Example3_UW
Example3_UW PROC NEAR
   ; exception/unwind handler body

   ret 0

Example3_UW ENDP

Example3 PROC FRAME : Example3_UW

   sub rsp, 16
.allocstack 16

.endprolog

   ; function body
    add rsp, 16
   ret 0

Example3 ENDP
text ENDS
END
```

See also

Directives Reference

MASM BNF Grammar

.ALPHA (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Orders segments alphabetically. (32-bit MASM only.)

Syntax

```
.ALPHA
```

See also

Directives Reference
MASM BNF Grammar

ASSUME

12/20/2019 • 2 minutes to read • Edit Online

Enables error checking for register values. (32-bit MASM only.)

Syntax

ASSUME *segregister*:*name* □, *segregister*:*name*...□
ASSUME *dataregister*:*type* □, *dataregister*:*type*...□
ASSUME *register*:ERROR □, *register*:ERROR...□
ASSUME □*register*:□NOTHING □, *register*:NOTHING...□

Remarks

After an **ASSUME** is put into effect, the assembler watches for changes to the values of the given registers. **ERROR** generates an error if the register is used. **NOTHING** removes register error checking. You can combine different kinds of assumptions in one statement.

See also

Directives Reference
MASM BNF Grammar

.BREAK (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates code to terminate a .WHILE or .REPEAT block if *condition* is true. (32-bit MASM only.)

Syntax

```
.BREAK .IF condition
```

See also

Directives Reference
MASM BNF Grammar

BYTE

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a byte of storage for each *initializer*.

Syntax

> □*name*□ **BYTE** *initializer* □, *initializer* ...□

Remarks

Can also be used as a type specifier anywhere a type is legal.

See also

Directives Reference
DB
SBYTE
MASM BNF Grammar

CATSTR

12/20/2019 • 2 minutes to read • Edit Online

Concatenates text items.

Syntax

> *name* **CATSTR** □*textitem1* □, *textitem2* ...□□

Remarks

Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

CATSTR is a synonym for TEXTEQU.

See also

Directives Reference
MASM BNF Grammar

.CODE

3/19/2020 • 2 minutes to read • Edit Online

(32-bit MASM only.) When used with .MODEL, indicates the start of a code segment.

Syntax

```
.CODE □name□
□ segmentItem □...
□ codesegmentnameId ENDS ;;□\
```

Parameters

name
Optional parameter that specifies the name of the code segment. The default name is **_TEXT** for tiny, small, compact, and flat models. The default name is *modulename*_TEXT for other models.

See also

Directives Reference
.DATA
MASM BNF Grammar

COMM

12/20/2019 • 2 minutes to read • Edit Online

Creates a communal variable with the attributes specified in *definition*.

Syntax

COMM *definition* □, *definition* ...□

Remarks

Communal variables are allocated by the linker, and can't be initialized. This means that you can't depend on the location or sequence of such variables.

Each *definition* has the following form:

□*language-type*□ □**NEAR** | **FAR**□ *label*:*type*□:*count*□

The *language-type*, **NEAR**, and **FAR** arguments are valid only in 32-bit MASM.

The optional *language-type* sets the naming conventions for the name that follows. It overrides any language specified by the **.MODEL** directive. The optional **NEAR** or **FAR** override the current memory model. The *label* is the name of the variable. The *type* can be any type specifier (BYTE, WORD, and so on) or an integer specifying the number of bytes. The optional *count* specifies the number of elements in the declared data object. The default *count* is one.

Example

This example creates an array of 512 BYTE elements:

```
COMM FAR ByteArray:BYTE:512
```

See also

Directives reference
MASM BNF Grammar

COMMENT

12/20/2019 • 2 minutes to read • Edit Online

Treats all *text* between or on the same line as the delimiters as a comment.

Syntax

> **COMMENT** *delimiter* □*text*□
> □*text*□
> □*text*□ *delimiter* □*text*□

See also

Directives Reference
MASM BNF Grammar

.CONST (32-bit MASM)

5/5/2020 • 2 minutes to read • Edit Online

When used with .MODEL, starts a constant data segment (with segment name **CONST**).

Syntax

```
.CONST
 segmentItem ...
```

Remarks

This segment has the read-only attribute.

See also

Directives Reference
MASM BNF Grammar

.CONTINUE (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates code to jump to the top of a .WHILE or .REPEAT block if *condition* is true. (32-bit MASM only.)

Syntax

```
.CONTINUE .IF condition
```

See also

Directives Reference
MASM BNF Grammar

.CREF

12/20/2019 • 2 minutes to read • Edit Online

Enables listing of symbols in the symbol portion of the symbol table and browser file.

Syntax

```
.CREF
```

See also

Directives Reference
MASM BNF Grammar

.DATA

3/19/2020 • 2 minutes to read • Edit Online

(32-bit MASM only.) When used with .MODEL, starts a near data segment for initialized data (segment name _DATA).

Syntax

> **.DATA**
> ⟦ *segmentItem* ⟧...

See also

Directives Reference
.DATA?
.CONST
.FARDATA
.FARDATA?
MASM BNF Grammar

.DATA?

12/20/2019 • 2 minutes to read • Edit Online

(32-bit MASM only.) When used with .MODEL, starts a near data segment for uninitialized data (segment name _BSS).

Syntax

```
.DATA?
 segmentItem ...
```

See also

Directives Reference
MASM BNF Grammar

DB

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a byte of storage for each *initializer*. **DB** is a synonym of BYTE.

Syntax

```
□name□ DB initializer □, initializer ...□
```

Remarks

Can also be used as a type specifier anywhere a type is legal.

See also

Directives Reference
DB
SBYTE
MASM BNF Grammar

DD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a double word (4 bytes) of storage for each *initializer*. **DD** is a synonym of DWORD.

Syntax

⟦*name*⟧ **DD** *initializer* ⟦, *initializer* ...⟧

Remarks

Can also be used as a type specifier anywhere a type is legal.

See also

Directives Reference
MASM BNF Grammar

DF

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 6 bytes of storage for each *initializer*. **DF** is a synonym of FWORD.

Syntax

> □*name*□ **DF** *initializer* □, *initializer* ...□

Remarks

Also can be used as a type specifier anywhere a type is legal.

See also

Directives Reference
MASM BNF Grammar

.DOSSEG (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Orders the segments according to the MS-DOS segment convention: CODE first, then segments not in DGROUP, and then segments in DGROUP. (32-bit MASM only.)

Syntax

```
.DOSSEG
```

Remarks

The segments in DGROUP follow this order: segments not in BSS or STACK, then BSS segments, and finally STACK segments. Primarily used for ensuring CodeView support in MASM stand-alone programs. Same as DOSSEG.

See also

Directives Reference
MASM BNF Grammar

DOSSEG

12/20/2019 • 2 minutes to read • Edit Online

(32-bit MASM only.) Identical to .DOSSEG, which is the preferred form.

Syntax

```
DOSSEG
```

See also

Directives Reference
MASM BNF Grammar

DQ

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 8 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal. **DQ** is a synonym of QWORD.

Syntax

> □*name*□ **DQ** *initializer* □, *initializer* ...□

See also

Directives Reference
QWORD
MASM BNF Grammar

DT

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 10 bytes of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal. DT is a synonym of TBYTE.

Syntax

□*name*□ DT *initializer* □, *initializer* ...□

See also

Directives reference
MASM BNF Grammar

DW

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal. **DW** is a synonym of WORD.

Syntax

> □*name*□ **DW** *initializer* □, *initializer* ...□

See also

Directives Reference
SWORD
MASM BNF Grammar

DWORD

3/16/2020 • 2 minutes to read • Edit Online

Allocates and optionally initializes a double word (4 bytes) of storage for each *initializer*. **DWORD** is a synonym of DD.

Syntax

> ⟦*name*⟧ **DWORD** *initializer* ⟦, *initializer* ...⟧

Remarks

Can also be used as a type specifier anywhere a type is legal.

See also

Directives Reference
SDWORD
DD
MASM BNF Grammar

ECHO

12/20/2019 • 2 minutes to read • Edit Online

Displays *message* to the standard output device (by default, the screen).

Syntax

```
ECHO message
```

Remarks

Same as %OUT.

See also

Directives reference
MASM BNF Grammar

.ELSE (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
.ELSE
```

Remarks

(32-bit MASM only.) See .IF.

See also

Directives Reference
MASM BNF Grammar

ELSE (MASM)

3/16/2020 • 2 minutes to read • Edit Online

Marks the beginning of an alternate block within a conditional block.

Syntax

```
ELSE
```

Remarks

See IF.

See also

Directives reference
MASM BNF Grammar

ELSEIF

12/20/2019 • 2 minutes to read • Edit Online

Combines ELSE and IF into one statement.

Syntax

ELSEIF *constantExpression*
statements
⟦**ELSE**
else-statements⟧
ENDIF

Remarks

See IF for more information.

See also

Directives reference
MASM BNF Grammar

ELSEIF2

12/20/2019 • 2 minutes to read • Edit Online

ELSEIF block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

Syntax

```
ELSEIF2
```

Remarks

For more information about 2-pass behavior in MASM 5.1 vs MASM 6.1, see IF1 and IF2.

See also

Directives reference
MASM BNF Grammar

END

12/20/2019 • 2 minutes to read • Edit Online

Marks the end of a module and, optionally, sets the program entry point to *procId*.

Syntax

```
END ⟦procId⟧
```

Remarks

The *procId* argument is valid in 32-bit MASM only.

See also

Directives reference
MASM BNF Grammar

.ENDIF (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
.ENDIF
```

Remarks

(32-bit MASM only.) See .IF.

See also

Directives Reference
MASM BNF Grammar

ENDM

12/20/2019 • 2 minutes to read • Edit Online

Terminates a macro or repeat block.

Syntax

```
ENDM
```

Remarks

See MACRO, FOR, FORC, REPEAT, or WHILE.

See also

Directives reference
MASM BNF Grammar

ENDP

12/20/2019 • 2 minutes to read • Edit Online

Marks the end of procedure *name* previously begun with **PROC**.

Syntax

> *name* **ENDP**

Remarks

See PROC.

See also

Directives reference
MASM BNF Grammar

.ENDPROLOG

12/20/2019 • 2 minutes to read • Edit Online

Signals the end of the prologue declarations.

Syntax

```
.ENDPROLOG
```

Remarks

It is an error to use any of the prologue declarations outside of the region between PROC **FRAME** and **.ENDPROLOG**.

For more information, see MASM for x64 (ml64.exe).

See also

Directives Reference
MASM BNF Grammar

ENDS

12/20/2019 • 2 minutes to read • Edit Online

Marks the end of segment, structure, or union *name* previously begun with SEGMENT, STRUCT, UNION, or a simplified segment directive.

Syntax

```
name ENDS
```

See also

Directives reference
MASM BNF Grammar

.ENDW (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
.ENDW
```

Remarks

(32-bit MASM only.) See .WHILE.

See also

Directives Reference
MASM BNF Grammar

EQU

12/20/2019 • 2 minutes to read • Edit Online

The first directive assigns numeric value of *expression* to *name*.

Syntax

> *name* **EQU** *expression*
>
> *name* **EQU** < *text*>

Remarks

The *name* cannot be redefined later.

The second directive assigns specified *text* to *name*. The *name* can be assigned a different *text* later. See TEXTEQU.

See also

Directives reference
MASM BNF Grammar

.ERR

9/21/2020 • 2 minutes to read • Edit Online

Generates an error.

Syntax

> `.ERR` 〚*message*〛

See also

Directives Reference
`.ERRNB`
`.ERR2`
`.ERRDEF`
`.ERRNZ`
MASM BNF Grammar

.ERR2

12/20/2019 • 2 minutes to read • Edit Online

.ERR block evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

Syntax

```
.ERR2 ⟦message⟧
```

Remarks

For more information about 2-pass behavior in MASM 5.1 vs MASM 6.1, see IF1 and IF2.

See also

Directives Reference
MASM BNF Grammar

.ERRB

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if *textitem* is blank.

Syntax

> .ERRB *textitem* □, *message*□

See also

Directives Reference
MASM BNF Grammar

.ERRDEF

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if *name* is a previously defined label, variable, or symbol.

Syntax

```
.ERRDEF name □, message□
```

See also

Directives Reference
MASM BNF Grammar

.ERRDIF, .ERRDIFI

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if the text items are different.

Syntax

> .ERRDIF *textitem1*, *textitem2* □, *message*□
> .ERRDIFI *textitem1*, *textitem2* □, *message*□

Remarks

If **.ERRDIFI** is given, the comparison is case insensitive.

See also

Directives reference
MASM BNF Grammar

.ERRE

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if *expression* is false (0).

Syntax

```
.ERRE expression ⟦, message⟧
```

See also

Directives Reference
MASM BNF Grammar

.ERRIDN, .ERRIDNI

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if the text items are identical.

Syntax

> **.ERRIDN** *textitem1*, *textitem2* □, *message*□
> **.ERRIDNI** *textitem1*, *textitem2* □, *message*□

Remarks

If **.ERRIDNI** is given, the comparison is case insensitive.

See also

Directives reference
MASM BNF Grammar

.ERRNB

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if *textitem* is not blank.

Syntax

> .ERRNB *textitem* □, *message*□

See also

Directives Reference
MASM BNF Grammar

.ERRNDEF

12/20/2019 • 2 minutes to read • Edit Online

Generates an error if *name* has not been defined.

Syntax

```
.ERRNDEF name □, message□
```

See also

Directives Reference
MASM BNF Grammar

.ERRNZ

9/21/2020 • 2 minutes to read • Edit Online

Generates an error if *expression* is true (nonzero).

Syntax

`.ERRNZ` *expression* ⟦`,` *message*⟧

See also

Directives Reference
MASM BNF Grammar

EVEN

12/20/2019 • 2 minutes to read • Edit Online

Aligns the next variable or instruction on an even byte.

Syntax

```
EVEN
```

See also

Directives reference
MASM BNF Grammar

.EXIT (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates termination code. (32-bit MASM only.)

Syntax

.EXIT ⟦*expression*⟧

Remarks

Returns optional *expression* to shell.

See also

Directives reference
MASM BNF Grammar

EXITM

12/20/2019 • 2 minutes to read • Edit Online

Terminates expansion of the current repeat or macro block and begins assembly of the next statement outside the block.

Syntax

> **EXITM** ⟦*textitem*⟧

Remarks

In a macro function, *textitem* is the value returned.

See also

Directives reference
MASM BNF Grammar

EXTERN

3/16/2020 • 2 minutes to read • Edit Online

Defines one or more external variables, labels, or symbols called *name* whose type is *type*.

Syntax

> **EXTERN** *language-type* *name* (*altid*) : *type* , *language-type* *name* (*altid*) : *type* ...

Remarks

The *language-type* argument is valid in 32-bit MASM only.

The *type* can be ABS, which imports *name* as a constant. Same as EXTRN.

See also

Directives Reference
MASM BNF Grammar

EXTERNDEF

12/20/2019 • 2 minutes to read • Edit Online

Defines one or more external variables, labels, or symbols called *name* whose type is *type*.

Syntax

> EXTERNDEF ⟦*language-type*⟧ *name*:*type* ⟦, ⟦*language-type*⟧ *name*:*type* ...⟧

Remarks

The *language-type* argument is valid in 32-bit MASM only.

If *name* is defined in the module, it is treated as PUBLIC. If *name* is referenced in the module, it is treated as EXTERN. If *name* is not referenced, it is ignored. The *type* can be ABS, which imports *name* as a constant. Normally used in include files.

See also

Directives Reference
MASM BNF Grammar

EXTRN

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
EXTRN
```

Remarks

See EXTERN.

See also

Directives reference
MASM BNF Grammar

.FARDATA (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

When used with .MODEL, starts a far data segment for initialized data (segment name FAR_DATA or *name*). (32-bit MASM only.)

Syntax

```
.FARDATA name
```

See also

Directives reference
MASM BNF Grammar

.FARDATA? (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

When used with .MODEL, starts a far data segment for uninitialized data (segment name FAR_BSS or *name*). (32-bit MASM only.)

Syntax

> .FARDATA? *name*

See also

Directives reference
MASM BNF Grammar

FOR

3/16/2020 • 2 minutes to read • Edit Online

Marks a block that will be repeated once for each *argument*, with the current *argument* replacing *parameter* on each repetition.

Syntax

> FOR *parameter* □:REQ | := *default*□ , < *argument* □, *argument* ...□>
> *statements*
> ENDM

Remarks

Same as IRP.

See also

Directives reference
MASM BNF Grammar

FORC

12/20/2019 • 2 minutes to read • Edit Online

Marks a block that will be repeated once for each character in *string*, with the current character replacing *parameter* on each repetition.

Syntax

> FORC *parameter*, < *string*>
> *statements*
> ENDM

Remarks

Same as IRPC.

See also

Directives reference
MASM BNF Grammar

.FPO (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

The **.FPO** directive controls the emission of debug records to the .debug$F segment or section. (32-bit MASM only.)

Syntax

> **.FPO** (*cdwLocals*, *cdwParams*, *cbProlog*, *cbRegs*, *fUseBP*, *cbFrame*)

Parameters

cdwLocals
Number of local variables, an unsigned 32 bit value.

cdwParams
Size of the parameters in DWORDS, an unsigned 16 bit value.

cbProlog
Number of bytes in the function prolog code, an unsigned 8 bit value.

cbRegs
Number registers saved.

fUseBP
Indicates whether the EBP register has been allocated. either 0 or 1.

cbFrame
Indicates the frame type. See FPO_DATA for more information.

See also

Directives reference
MASM BNF Grammar

FWORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 6 bytes of storage for each *initializer*.

Syntax

> □*name*□ **FWORD** *initializer* □, *initializer* ...□

Remarks

Also can be used as a type specifier anywhere a type is legal.

See also

Directives Reference
DF
MASM BNF Grammar

GOTO

3/16/2020 • 2 minutes to read • Edit Online

Transfers assembly to the line marked :*macrolabel*.

Syntax

> GOTO *macrolabel*

Remarks

GOTO is permitted only inside MACRO, FOR, FORC, REPEAT, and WHILE blocks. The *macrolabel* target must be the only directive on the line and must be preceded by a leading colon.

See also

Directives reference
MASM BNF Grammar

GROUP

12/20/2019 • 2 minutes to read • Edit Online

(32-bit MASM only.) Add the specified *segments* to the group called *name*.

Syntax

> *name* **GROUP** *segment* ⟦, *segment* ...⟧

See also

Directives reference
MASM BNF Grammar

.IF (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates code that tests *condition1* (for example, AX > 7) and executes the *statements* if that condition is true. (32-bit MASM only.)

Syntax

```
.IF condition1
statements
.ELSEIF condition2
statements
.ELSE
statements
.ENDIF
```

Remarks

If a .ELSE follows, its statements are executed if the original condition was false. Note that the conditions are evaluated at run time.

See also

Directives reference
MASM BNF Grammar

IF

3/16/2020 • 2 minutes to read • Edit Online

Grants assembly of *ifstatements* if *expression1* is true (nonzero) or *elseifstatements* if *expression1* is false (0) and *expression2* is true.

Syntax

> **IF** *expression1*
> *if-statements*
> **ELSEIF** *expression2*
> *elseif-statements*
> **ELSE**
> *else-statements*
> **ENDIF**

Remarks

The following directives may be substituted for ELSEIF: **ELSEIFB**, **ELSEIFDEF**, **ELSEIFDIF**, **ELSEIFDIFI**, **ELSEIFE**, **ELSEIFIDN**, **ELSEIFIDNI**, **ELSEIFNB**, and **ELSEIFNDEF**. Optionally, assembles *else-statements* if the previous expression is false. Note that the expressions are evaluated at assembly time.

See also

Directives reference
MASM BNF Grammar

IF1 and IF2

12/20/2019 • 2 minutes to read • Edit Online

IF1 block is evaluated on first assembly pass.

IF2 block is evaluated on every assembly pass if **OPTION:SETIF2** is **TRUE**.

Syntax

IF1;;

IF2;;

Remarks

See IF for complete syntax.

Unlike version 5.1, MASM 6.1 and above do most of its work on its first pass, then performs as many subsequent passes as necessary. In contrast, MASM 5.1 always assembles in two source passes. As a result, you may need to revise or delete some pass-dependent constructs under MASM 6.1 and above.

Two-Pass Directives

To assure compatibility, MASM 6.1 and above support 5.1 directives referring to two passes. These include **.ERR1**, **.ERR2**, **IF1**, **IF2**, **ELSEIF1**, and **ELSEIF2**. For second-pass constructs, you must specify OPTION SETIF2. Without **OPTION SETIF2**, the **IF2** and **.ERR2** directives cause an error:

```
.ERR2 not allowed : single-pass assembler
```

MASM 6.1 and above handle first-pass constructs differently. It treats the **.ERR1** directive as **.ERR**, and the **IF1** directive as **IF**.

See also

Directives reference
MASM BNF Grammar

IFB

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if *textitem* is blank.

Syntax

```
IFB textitem
```

Remarks

See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFDEF

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if *name* is a previously defined label, variable, or symbol.

Syntax

> **IFDEF** *name*

Remarks

See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFDIF, IFDIFI

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if the text items are different.

Syntax

> **IFDIF** *textitem1*, *textitem2*
> **IFDIFI** *textitem1*, *textitem2*

Remarks

If **IFDIFI** is given, the comparison is case insensitive. See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFE

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if *expression* is false (0).

Syntax

```
IFE expression
```

Remarks

See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFIDN, IFIDNI

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if the text items are identical.

Syntax

> **IFIDN** *textitem1*, *textitem2*
> **IFIDNI** *textitem1*, *textitem2*

Remarks

If **IFIDNI** is given, the comparison is case insensitive. See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFNB

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if *textitem* is not blank.

Syntax

```
IFNB textitem
```

Remarks

See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

IFNDEF

12/20/2019 • 2 minutes to read • Edit Online

Grants assembly if *name* has not been defined.

Syntax

```
IFNDEF name
```

Remarks

See IF for complete syntax.

See also

Directives reference
MASM BNF Grammar

INCLUDE

3/16/2020 • 2 minutes to read • Edit Online

Inserts source code from the source file given by *filename* into the current source file during assembly.

Syntax

```
INCLUDE filename
```

Remarks

The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

See also

Directives reference
MASM BNF Grammar

INCLUDELIB

12/20/2019 • 2 minutes to read • Edit Online

Informs the linker that the current module should be linked with *libraryname*.

Syntax

> **INCLUDELIB** *libraryname*

Remarks

The *libraryname* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.

See also

Directives reference
MASM BNF Grammar

INSTR

12/20/2019 • 2 minutes to read • Edit Online

Finds the first occurrence of *textitem2* in *textitem1*.

Syntax

> *name* **INSTR** ⟦*position*,⟧ *textitem1*, *textitem2*

Remarks

The starting *position* is optional. Each text item can be a literal string, a constant preceded by a **%**, or the string returned by a macro function.

See also

Directives reference
MASM BNF Grammar

INVOKE

3/19/2020 • 2 minutes to read • Edit Online

(32-bit MASM only.) Calls the procedure at the address given by *expression*, passing the arguments on the stack or in registers according to the standard calling conventions of the language type.

Syntax

> **INVOKE** *expression* ⟦, *argument* ...⟧

Remarks

Each argument passed to the procedure may be an expression, a register pair, or an address expression (an expression preceded by **ADDR**).

See also

Directives reference
MASM BNF Grammar

IRP

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
IRP
```

Remarks

See FOR.

See also

Directives reference
MASM BNF Grammar

IRPC

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
IRPC
```

Remarks

See FORC.

See also

Directives reference
MASM BNF Grammar

.K3D (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of **K3D** instructions. (32-bit MASM only.)

Syntax

```
.K3D
```

See also

Directives reference
MASM BNF Grammar

LABEL

12/20/2019 • 2 minutes to read • Edit Online

Creates a new label by assigning the current location-counter value and the given *qualifiedType* to *name*.

Syntax

> *name* **LABEL** *qualifiedType*
>
> *name* **LABEL ⟦NEAR | FAR | PROC⟧ PTR ⟦***qualifiedType***⟧**

See also

Directives reference
MASM BNF Grammar

.LALL

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
.LALL
```

Remarks

See .LISTMACROALL.

See also

Directives reference
MASM BNF Grammar

.LFCOND

12/20/2019 • 2 minutes to read • Edit Online

Syntax

```
.LFCOND
```

Remarks

See .LISTIF.

See also

Directives reference
MASM BNF Grammar

.LIST

12/20/2019 • 2 minutes to read • Edit Online

Starts listing of statements.

Syntax

```
.LIST
```

Remarks

This is the default.

See also

Directives reference
MASM BNF Grammar

.LISTALL

12/20/2019 • 2 minutes to read • Edit Online

Starts listing of all statements.

Syntax

```
.LISTALL
```

Remarks

Equivalent to the combination of .LIST, .LISTIF, and .LISTMACROALL.

See also

Directives reference
MASM BNF Grammar

.LISTIF

12/20/2019 • 2 minutes to read • Edit Online

Starts listing of statements in false conditional blocks.

Syntax

```
.LISTIF
```

Remarks

Same as .LFCOND.

See also

Directives reference
MASM BNF Grammar

.LISTMACRO

12/20/2019 • 2 minutes to read • Edit Online

Starts listing of macro expansion statements that generate code or data.

Syntax

```
.LISTMACRO
```

Remarks

This is the default. Same as .XALL.

See also

Directives reference
MASM BNF Grammar

.LISTMACROALL

12/20/2019 • 2 minutes to read • Edit Online

Starts listing of all statements in macros.

Syntax

```
.LISTMACROALL
```

Remarks

Same as .LALL.

See also

Directives reference
MASM BNF Grammar

LOCAL

12/20/2019 • 2 minutes to read • Edit Online

In the first directive, within a macro, **LOCAL** defines labels that are unique to each instance of the macro.

Syntax

> **LOCAL** *localId* □, *localId* ...□
>
> **LOCAL** *labelId* □ [*count*] □ □:*qualifiedType*□ □, *labelId* □ [*count*] □ □*qualifiedType*□ ...□

Remarks

In the second directive, within a procedure definition (**PROC**), **LOCAL** creates stack-based variables that exist for the duration of the procedure. The *labelId* may be a simple variable or an array containing *count* elements, where *count* is a constant expression.

See also

Directives reference
MASM BNF Grammar

MACRO

3/19/2020 • 2 minutes to read • Edit Online

Marks a macro block called *name* and establishes *parameter* placeholders for arguments passed when the macro is called.

Syntax

name **MACRO** □*parameter* □**:REQ** | **:=***default* | *args* **:VARARG**□ ...□
statements
□**GOTO** **:***macrolabelId*□
□**EXITM**□
ENDM □*value*□

Remarks

A macro function returns *value* to the calling statement.

See also

Directives reference
GOTO (MASM)
ENDM
MASM BNF Grammar

MMWORD

12/20/2019 • 2 minutes to read • Edit Online

Used for 64-bit multimedia operands with MMX and SSE (XMM) instructions.

Syntax

> **MMWORD**

Remarks

MMWORD is a type. Prior to **MMWORD** being added to MASM, equivalent functionality could have been achieved with:

```
mov mm0, qword ptr [ebx]
```

While both instructions work on 64-bit operands, **QWORD** is the type for 64-bit unsigned integers and **MMWORD** is the type for a 64-bit multimedia value.

MMWORD is intended to represent the same type as __m64.

Example

```
movq     mm0, mmword ptr [ebx]
```

See Also

MASM BNF Grammar

.MMX (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of MMX or single-instruction, multiple data (SIMD) instructions. (32-bit MASM only.)

Syntax

```
.MMX
```

See also

Directives reference
MASM BNF Grammar

.MODEL (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Initializes the program memory model. (32-bit MASM only.)

Syntax

> .MODEL *memory-model* □, *language-type*□ □, *stack-option*□

Parameters

memory-model
Required parameter that determines the size of code and data pointers.

language-type
Optional parameter that sets the calling and naming conventions for procedures and public symbols.

stack-option
Optional parameter.

stack-option is not used if *memory-model* is **FLAT**.

Specifying **NEARSTACK** groups the stack segment into a single physical segment (**DGROUP**) along with data. The stack segment register (**SS**) is assumed to hold the same address as the data segment register (**DS**). **FARSTACK** does not group the stack with **DGROUP**; thus **SS** does not equal **DS**.

Remarks

.MODEL is not used in MASM for x64 (ml64.exe).

The following table lists the possible values for each parameter when targeting 16-bit and 32-bit platforms:

PARAMETER	32-BIT VALUES	16-BIT VALUES (SUPPORT FOR EARLIER 16-BIT DEVELOPMENT)
memory-model	**FLAT**	**TINY**, **SMALL**, **COMPACT**, **MEDIUM**, **LARGE**, **HUGE**, **FLAT**
language-type	**C**, **STDCALL**	**C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, **STDCALL**
stack-option	Not used	**NEARSTACK**, **FARSTACK**

Code

For MASM-related samples, download the Compiler samples from Visual C++ Samples and Related Documentation for Visual Studio 2010.

The following example demonstrates the use of the `.MODEL` directive.

Example

```
; file simple.asm
; For x86 (32-bit), assemble with debug information:
;   ml -c -Zi simple.asm
; For x64 (64-bit), assemble with debug information:
;   ml64 -c -DX64 -Zi simple.asm
;
; In this sample, the 'X64' define excludes source not used
;  when targeting the x64 architecture

ifndef X64
.686p
.XMM
.model flat, C
endif

.data
; user data

.code
; user code

fxn PROC public
  xor eax, eax ; zero function return value
  ret
fxn ENDP

end
```

See also

Directives Reference
MASM BNF Grammar

NAME

3/16/2020 • 2 minutes to read • Edit Online

Ignored.

Syntax

```
NAME
```

See also

Directives reference
MASM BNF Grammar

.NOCREF

12/20/2019 • 2 minutes to read • Edit Online

Suppresses listing of symbols in the symbol table and browser file.

Syntax

```
.NOCREF □name□, name ...□□
```

Remarks

If names are specified, then only the given names are suppressed. Same as .XCREF.

See also

Directives Reference
MASM BNF Grammar

.NOLIST

12/20/2019 • 2 minutes to read • Edit Online

Suppresses program listing.

Syntax

```
.NOLIST
```

Remarks

Same as .XLIST.

See also

Directives reference
MASM BNF Grammar

.NOLISTIF

12/20/2019 • 2 minutes to read • Edit Online

Suppresses listing of conditional blocks whose condition evaluates to false (0).

Syntax

```
.NOLISTIF
```

Remarks

This is the default. Same as .SFCOND.

See also

Directives reference MASM BNF Grammar

.NOLISTMACRO

12/20/2019 • 2 minutes to read • Edit Online

Suppresses listing of macro expansions.

Syntax

```
.NOLISTMACRO
```

Remarks

Same as .SALL.

See also

Directives reference
MASM BNF Grammar

OPTION

9/21/2020 • 2 minutes to read • Edit Online

Enables and disables features of the assembler.

Syntax

> `OPTION` *option-list*

Remarks

Available options include:

`CASEMAP`
`DOTNAME`
`NODOTNAME`
`EMULATOR`
`NOEMULATOR`
`EPILOGUE`
`EXPR16`

`EXPR32`
`LANGUAGE`
`LJMP`
`NOLJMP`
`M510`
`NOM510`
`NOKEYWORD`

`NOSIGNEXTEND`
`OFFSET`
`OLDMACROS`
`NOOLDMACROS`
`OLDSTRUCTS`
`NOOLDSTRUCTS`
`PROC`

`PROLOGUE`
`READONLY`
`NOREADONLY`
`SCOPED`
`NOSCOPED`
`SEGMENT`
`SETIF2`

The syntax for LANGUAGE is `OPTION LANGUAGE:` *`x`*, where *`x`* is one of `C`, `SYSCALL`, `STDCALL`, `PASCAL`, `FORTRAN`, or `BASIC`. `SYSCALL`, `PASCAL`, `FORTRAN`, and `BASIC` are not supported with `.MODEL` `FLAT`.

See also

Directives reference
MASM BNF Grammar

ORG

12/20/2019 • 2 minutes to read • Edit Online

Sets the location counter to *expression*.

Syntax

```
ORG expression
```

See also

Directives reference
MASM BNF Grammar

%OUT

12/20/2019 • 2 minutes to read • Edit Online

See ECHO.

Syntax

```
%OUT
```

See also

Directives reference
MASM BNF Grammar

OWORD

12/20/2019 • 2 minutes to read • Edit Online

Used as a type specifier when an 16-byte data type is required.

Syntax

> ⟦*name*⟧ **OWORD** *initializer* ⟦, *initializer* ...⟧

See also

Directives reference
MASM BNF Grammar

PAGE

12/20/2019 • 2 minutes to read • Edit Online

The first directive sets line *length* and character *width* of the program listing. If no arguments are given, generates a page break. The second directive increments the section number and resets the page number to 1.

Syntax

```
PAGE ⟦length⟧⟦, width⟧
```

```
PAGE +
```

See also

Directives reference
MASM BNF Grammar

POPCONTEXT

12/20/2019 • 2 minutes to read • Edit Online

Restores part or all of the current *context* (saved by the PUSHCONTEXT directive). The *context* can be **ASSUMES** (32-bit MASSM only), **RADIX**, **LISTING**, **CPU** (32-bit MASSM only), or **ALL**.

Syntax

> **POPCONTEXT** *context*

See also

Directives reference
MASM BNF Grammar

PROC

12/20/2019 • 2 minutes to read • Edit Online

Marks start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction or INVOKE directive.

Syntax

label **PROC** □*distance*□ □*language-type*□ □ **PUBLIC** | **PRIVATE** | **EXPORT** □ □< *prologuearg*> □ □**USES** *reglist*□ □, *parameter* □:*tag*□ ...□
□**FRAME** □:*ehandler-address*□ □
statements
label **ENDP**

Remarks

The □*distance*□ and □*language-type*□ arguments are valid only in 32-bit MASM.

□**FRAME** □:*ehandler-address*□ □ is only valid with ml64.exe, and causes MASM to generate a function table entry in .pdata and unwind information in .xdata for a function's structured exception handling unwind behavior.

When the **FRAME** attribute is used, it must be followed by an .ENDPROLOG directive.

See MASM for x64 (ml64.exe) for more information on using ml64.exe.

Example

```
; ml64 ex1.asm /link /entry:Example1 /SUBSYSTEM:CONSOLE
_text SEGMENT
Example1 PROC FRAME
   push r10
.pushreg r10
   push r15
.pushreg r15
   push rbx
.pushreg rbx
   push rsi
.pushreg rsi
.endprolog
   ; rest of function ...
   ret
Example1 ENDP
_text ENDS
END
```

The above code will emit the following function table and unwind information:

```
FileHeader->Machine 34404
Dumping Unwind Information for file ex2.exe

.pdata entry 1 0x00001000 0x00001023

  Unwind data: 0x00002000

    Unwind version: 1
    Unwind Flags: None
    Size of prologue: 0x08
    Count of codes: 3
    Frame register: rbp
    Frame offset: 0x0
    Unwind codes:

      Code offset: 0x08, SET_FPREG, register=rbp, offset=0x00
      Code offset: 0x05, ALLOC_SMALL, size=0x10
      Code offset: 0x01, PUSH_NONVOL, register=rbp
```

See also

Directives reference
MASM BNF Grammar

PROTO

12/20/2019 • 2 minutes to read • Edit Online

Prototypes a function or procedure. You can call the function prototyped by the PROTO directive by using the INVOKE directive.

Syntax

> *label* **PROTO** □*distance*□ □*language-type*□ □, □*parameter*□**:***tag* ...□

Parameters

label
The name of the prototyped function.

distance (32-bit MASM only.)
(Optional) Used in 16-bit memory models to override the default and indicate **NEAR** or **FAR** calls.

language-type (32-bit MASM only.)
(Optional) Sets the calling and naming convention for procedures and public symbols. Supported conventions are:

- 32-bit **FLAT** model: **C**, **STDCALL**
- 16-bit models: **C**, **BASIC**, **FORTRAN**, **PASCAL**, **SYSCALL**, **STDCALL**

parameter
The optional name for a function parameter.

tag
The type of a function parameter.

The *parameter* and *tag* parameters may appear multiple times, once for each passed argument.

Example

This sample shows a **PROTO** declaration for a function named `addup3` that uses a **NEAR** call to override the 16-bit model default for procedure calls, and uses the **C** calling convention for stack parameters and return values. It takes two arguments, a **WORD** and a **VARARG**.

```
addup3 PROTO NEAR C, argcount:WORD, arg1:VARARG
```

See also

Directives Reference
.MODEL Reference
MASM BNF Grammar

PUBLIC

12/20/2019 • 2 minutes to read • Edit Online

Makes each variable, label, or absolute symbol specified as *name* available to all other modules in the program.

Syntax

> **PUBLIC** ⟦*language-type*⟧ *name* ⟦, ⟦*language-type*⟧ *name* ...⟧

Remarks

The *language-type* argument is valid in 32-bit MASM only.

See also

Directives reference
MASM BNF Grammar

PURGE

12/20/2019 • 2 minutes to read • Edit Online

Deletes the specified macros from memory.

Syntax

> PURGE *macronameId* , *macronameId* ...

See also

Directives reference MASM BNF Grammar

PUSHCONTEXT

12/20/2019 • 2 minutes to read • Edit Online

Saves part or all of the current *context*: segment register assumes, radix value, listing and cref flags, or processor/coprocessor values. The *context* can be **ASSUMES** (32-bit MASM only), **RADIX**, **LISTING**, **CPU** (32-bit MASM only), or **ALL**.

Syntax

> **PUSHCONTEXT** *context*

See also

Directives reference
MASM BNF Grammar

.PUSHFRAME

12/20/2019 • 2 minutes to read • Edit Online

Generates a `UWOP_PUSH_MACHFRAME` unwind code entry. If the optional **CODE** keyword is specified, the unwind code entry is given a modifier of 1. Otherwise the modifier is 0.

Syntax

```
.PUSHFRAME □CODE□;;
```

Remarks

.PUSHFRAME allows ml64.exe users to specify how a frame function unwinds. It's only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives don't generate code; they only generate `.xdata` and `.pdata`. **.PUSHFRAME** should be preceded by instructions that actually implement the actions to be unwound. It's a good practice to wrap both the unwind directives and the code they're meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

See also

Directives reference
MASM BNF Grammar

.PUSHREG

12/20/2019 • 2 minutes to read • Edit Online

Generates a `UWOP_PUSH_NONVOL` unwind code entry for the specified register number using the current offset in the prologue.

Syntax

.PUSHREG *register*

Remarks

.PUSHREG allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC **FRAME** declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. **.PUSHREG** should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

register may be one of:
RAX | RCX | RDX | RBX | RDI | RSI | RBP | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15.

For more information, see MASM for x64 (ml64.exe).

Sample

Description

The following sample shows how to push non-volatile registers.

Code

```
; ml64 ex1.asm /link /entry:Example1 /SUBSYSTEM:CONSOLE
_text SEGMENT
Example1 PROC FRAME
   push r10
.pushreg r10
   push r15
.pushreg r15
   push rbx
.pushreg rbx
   push rsi
.pushreg rsi
.endprolog
   ; rest of function ...
   ret
Example1 ENDP
_text ENDS
END
```

See also

Directives reference
MASM BNF Grammar

QWORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 8 bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal. **QWORD** is a synonym of DQ.

Syntax

> ⟦*name*⟧ **QWORD** *initializer* ⟦, *initializer* ...⟧

See also

Directives Reference
SQWORD
DQ
MASM BNF Grammar

.RADIX

12/20/2019 • 2 minutes to read • Edit Online

Sets the default radix, in the range 2 to 16, to the value of *expression*.

Syntax

```
.RADIX expression
```

See also

Directives reference
MASM BNF Grammar

REAL10

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a 10-byte floating-point number for each *initializer*.

Syntax

> *name* **REAL10** *initializer* □, *initializer* ...□

See also

Directives reference

MASM BNF Grammar

REAL4

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a single-precision (4-byte) floating-point number for each *initializer*.

Syntax

> *name* **REAL4** *initializer* □, *initializer* ...□

See also

Directives reference

MASM BNF Grammar

REAL8

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a double-precision (8-byte) floating-point number for each *initializer*.

Syntax

> *name* **REAL8** *initializer* ⟦, *initializer* ...⟧

See also

Directives reference
MASM BNF Grammar

RECORD

12/20/2019 • 2 minutes to read • Edit Online

Declares a record type consisting of the specified fields. *fieldname* names the field, *width* specifies the number of bits, and *expression* gives its initial value.

Syntax

> *recordname* **RECORD** *fieldname*:*width* ⟦= *expression*⟧ ⟦, *fieldname*:*width* ⟦= *expression*⟧ ...⟧

See also

Directives reference
MASM BNF Grammar

.REPEAT (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates code that repeats execution of the block of *statements* until *condition* becomes true. .UNTILCXZ, which becomes true when CX is zero, may be substituted for .UNTIL. The *condition* is optional with **.UNTILCXZ**. (32-bit MASM only.)

Syntax

> **.REPEAT**
> *statements*
> **.UNTIL** *condition*

See also

Directives reference
MASM BNF Grammar

REPEAT

12/20/2019 • 2 minutes to read • Edit Online

Marks a block that is to be repeated *expression* times. Same as REPT.

Syntax

> **REPEAT** *expression*
> *statements*
> **ENDM**

See also

Directives reference
MASM BNF Grammar

REPT

12/20/2019 • 2 minutes to read • Edit Online

See REPEAT.

Syntax

```
REPT
```

See also

Directives reference
MASM BNF Grammar

.SAFESEH (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Registers a function as a structured exception handler. (32-bit MASM only.)

Syntax

> .SAFESEH *identifier*

Remarks

identifier must be the ID for a locally defined PROC or EXTRN PROC. A LABEL is not allowed. The .SAFESEH directive requires the /safeseh ml.exe command-line option.

For more information about structured exception handlers, see /SAFESEH.

For example, to register a safe exception handler, create a new MASM file (as follows), assemble with /safeseh, and add it to the linked objects.

```
.386
.model  flat
MyHandler   proto
.safeseh    MyHandler
end
```

See also

Directives reference
MASM BNF Grammar

.SALL

12/20/2019 • 2 minutes to read • Edit Online

See .NOLISTMACRO.

Syntax

```
.SALL
```

See also

Directives reference
MASM BNF Grammar

.SAVEREG

12/20/2019 • 2 minutes to read • Edit Online

Generates either a `UWOP_SAVE_NONVOL` or a `UWOP_SAVE_NONVOL_FAR` unwind code entry for the specified register (*reg*) and offset (*offset*) using the current prologue offset. MASM will choose the most efficient encoding.

Syntax

> .SAVEREG *reg*, *offset*

Remarks

.SAVEREG allows ml64.exe users to specify how a frame function unwinds and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. **.SAVEREG** should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

See also

Directives reference
MASM BNF Grammar

.SAVEXMM128

12/20/2019 • 2 minutes to read • Edit Online

Generates either a `UWOP_SAVE_XMM128` or a `UWOP_SAVE_XMM128_FAR` unwind code entry for the specified XMM register and offset using the current prologue offset. MASM will choose the most efficient encoding.

Syntax

> .SAVEXMM128 *xmmreg* , *offset*

Remarks

.SAVEXMM128 allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .SAVEXMM128 should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

offset must be a multiple of 16.

For more information, see MASM for x64 (ml64.exe).

See also

Directives reference
MASM BNF Grammar

SBYTE

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a signed byte of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

Syntax

> *name* **SBYTE** *initializer* □, *initializer* ...□

See also

Directives Reference
BYTE
DB
MASM BNF Grammar

SDWORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a signed double word (4 bytes) of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

Syntax

> *name* **SDWORD** *initializer* □, *initializer* ...□

See also

Directives Reference
DWORD
DD
MASM BNF Grammar

SEGMENT

12/20/2019 • 2 minutes to read • Edit Online

Defines a program segment called *name* having segment attributes

Syntax

> *name* SEGMENT □READONLY□ □*align*□ □*combine*□ □*use*□ □*characteristics*□ ALIAS(*string*) □'*class*'□
> *statements*
> *name* ENDS

Parameters

align
The range of memory addresses from which a starting address for the segment can be selected. The alignment type can be any one of the following:

ALIGN TYPE	STARTING ADDRESS
BYTE	Next available byte address.
WORD	Next available word address (2 bytes per word).
DWORD	Next available double word address (4 bytes per double word).
PARA	Next available paragraph address (16 bytes per paragraph).
PAGE	Next available page address (256 bytes per page).
ALIGN(*n*)	Next available *n*th byte address. See Remarks section for more information.

If this parameter is not specified, **PARA** is used by default.

combine (32-bit MASM only)
PUBLIC, **STACK**, **COMMON**, **MEMORY**, **AT** *address*, **PRIVATE**

use (32-bit MASM only)
USE16, **USE32**, **FLAT**

characteristics
INFO, **READ**, **WRITE**, **EXECUTE**, **SHARED**, **NOPAGE**, **NOCACHE**, and **DISCARD**

These are supported for COFF only and correspond to the COFF section characteristics of similar name (for example, **SHARED** corresponds to IMAGE_SCN_MEM_SHARED). READ sets the IMAGE_SCN_MEM_READ flag. The obsolete READONLY flag caused the section to clear the IMG_SCN_MEM_WRITE flag. If any *characteristics* are set, the default characteristics are not used and only the programmer-specified flags are in effect.

string
This string is used as the section name in the emitted COFF object. Creates multiple sections with the same external name, with distinct MASM segment names.

Not supported with **/omf**.

class
Designates how segments should be combined and ordered in the assembled file. Typical values are, `'DATA'`, `'CODE'`, `'CONST'` and `'STACK'`

Remarks

For `ALIGN(n)`, *n* may be any power of 2 from 1 to 8192; not supported with **/omf**.

See also

Directives reference
MASM BNF Grammar

.SEQ (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Orders segments sequentially (the default order). (32-bit MASM only.)

Syntax

```
.SEQ
```

See also

Directives reference
MASM BNF Grammar

.SETFRAME

12/20/2019 • 2 minutes to read • Edit Online

Fills in the frame register field and offset in the unwind information using the specified register (*reg*) and offset (*offset*). The offset must be a multiple of 16 and less than or equal to 240. This directive also generates a `UWOP_SET_FPREG` unwind code entry for the specified register using the current prologue offset.

Syntax

> .SETFRAME *reg*, *offset*

Remarks

.SETFRAME allows ml64.exe users to specify how a frame function unwinds, and is only allowed within the prologue, which extends from the PROC FRAME declaration to the .ENDPROLOG directive. These directives do not generate code; they only generate `.xdata` and `.pdata`. .SETFRAME should be preceded by instructions that actually implement the actions to be unwound. It is a good practice to wrap both the unwind directives and the code they are meant to unwind in a macro to ensure agreement.

For more information, see MASM for x64 (ml64.exe).

Sample

Description

The following sample shows how to use a frame pointer:

Code

```
; ml64 frmex2.asm /link /entry:frmex2 /SUBSYSTEM:CONSOLE
_text SEGMENT
frmex2 PROC FRAME
   push rbp
.pushreg rbp
   sub rsp, 010h
.allocstack 010h
   mov rbp, rsp
.setframe rbp, 0
.endprolog
   ; modify the stack pointer outside of the prologue (similar to alloca)
   sub rsp, 060h

   ; we can unwind from the following AV because of the frame pointer
   mov rax, 0
   mov rax, [rax] ; AV!

   add rsp, 060h
   add rsp, 010h
   pop rbp
   ret
frmex2 ENDP
_text ENDS
END
```

See also

Directives reference

MASM BNF Grammar

.SFCOND

12/20/2019 • 2 minutes to read • Edit Online

See .NOLISTIF.

Syntax

```
.SFCOND
```

See also

Directives reference
MASM BNF Grammar

SIZESTR

12/20/2019 • 2 minutes to read • Edit Online

Finds the size of a text item.

Syntax

> *name* **SIZESTR** *textitem*

See also

Directives reference
MASM BNF Grammar

SQWORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 8 signed bytes of storage for each *initializer*. Also can be used as a type specifier anywhere a type is legal.

Syntax

> *name* **SQWORD** *initializer* ⟦, *initializer* ...⟧

See also

Directives Reference
QWORD
DQ
MASM BNF Grammar

.STACK (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

When used with .MODEL, defines a stack segment (with segment name **STACK**). The optional *size* specifies the number of bytes for the stack (default 1,024). The **.STACK** directive automatically closes the stack statement. (32-bit MASM only.)

Syntax

> .STACK [*size*]

See also

Directives reference
MASM BNF Grammar

.STARTUP (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates program start-up code. (32-bit MASM only.)

Syntax

```
.STARTUP
```

See also

Directives reference
MASM BNF Grammar

STRUC

12/20/2019 • 2 minutes to read • Edit Online

See STRUCT.

Syntax

```
STRUC
```

See also

Directives reference
MASM BNF Grammar

STRUCT

5/5/2020 • 2 minutes to read • Edit Online

Declares a structure type having the specified *field-declarations*. Each field must be a valid data definition. Same as STRUC.

Syntax

> *name* STRUCT ⟦*alignment*⟧ ⟦, NONUNIQUE⟧
> *field-declarations*
> *name* ENDS

Remarks

The *name* argument must be the same in the opening and closing statement.

See also

Directives reference
MASM BNF Grammar

SUBSTR

12/20/2019 • 2 minutes to read • Edit Online

Returns a substring of *textitem*, starting at *position*. The *textitem* can be a literal string, a constant preceded by a `%`, or the string returned by a macro function.

Syntax

> *name* **SUBSTR** *textitem*, *position* □, *length*□

See also

Directives reference
MASM BNF Grammar

SUBTITLE

12/20/2019 • 2 minutes to read • Edit Online

Defines the listing subtitle. Same as SUBTTL.

Syntax

> **SUBTITLE** *text*

See also

Directives reference
MASM BNF Grammar

SUBTTL

12/20/2019 • 2 minutes to read • Edit Online

See SUBTITLE.

Syntax

```
SUBTTL
```

See also

Directives reference MASM BNF Grammar

SWORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a signed word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal.

Syntax

> *name* **SWORD** *initializer* □, *initializer* ...□

See also

Directives reference
WORD
MASM BNF Grammar

TBYTE

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes 10 bytes of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal. **DT** is a synonym of **TBYTE**.

Syntax

> □*name*□ **TBYTE** *initializer* □, *initializer* ...□

See also

Directives Reference
DT
MASM BNF Grammar

TEXTEQU

12/20/2019 • 2 minutes to read • Edit Online

Assigns *textitem* to *name*. The *textitem* can be a literal string, a constant preceded by a `%`, or the string returned by a macro function.

Syntax

> *name* TEXTEQU 〚*textitem*〛

See also

Directives reference
MASM BNF Grammar

.TFCOND

12/20/2019 • 2 minutes to read • Edit Online

Toggles listing of false conditional blocks.

Syntax

```
.TFCOND
```

See also

Directives reference
MASM BNF Grammar

TITLE

12/20/2019 • 2 minutes to read • Edit Online

Defines the program listing title.

Syntax

```
TITLE text
```

See also

Directives reference
MASM BNF Grammar

TYPEDEF

12/20/2019 • 2 minutes to read • Edit Online

Defines a new type or **PROTO** called *name*, which is equivalent to *type* or *protoDefinition*.

Syntax

> *name* **TYPEDEF** *type* | **PROTO** *protoDefinition*

See also

Directives reference
MASM BNF Grammar

UNION

12/20/2019 • 2 minutes to read • Edit Online

Declares a union of one or more data types. The *field-declarations* must be valid data definitions. Omit the ENDS *name* label on nested **UNION** definitions.

Syntax

> *name* **UNION** ⟦*alignment*⟧ ⟦, **NONUNIQUE**⟧
> *field-declarations*
> ⟦*name*⟧ **ENDS**

See also

Directives reference
MASM BNF Grammar

.UNTIL (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

(32-bit MASM only.) See .REPEAT.

Syntax

```
.UNTIL
```

See also

Directives reference
MASM BNF Grammar

.UNTILCXZ (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

(32-bit MASM only.) See .REPEAT.

See also

Directives reference
MASM BNF Grammar

.WHILE (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Generates code that executes the block of *statements* while *condition* remains true. (32-bit MASM only.)

Syntax

> **.WHILE** *condition*
> *statements*
> **.ENDW**

See also

Directives reference
MASM BNF Grammar

WHILE

3/16/2020 • 2 minutes to read • Edit Online

Repeats assembly of block *statements* as long as *constantExpression* remains true.

Syntax

> **WHILE** *constantExpression*
> *statements*
> **ENDM**

See also

Directives reference
MASM BNF Grammar

WORD

12/20/2019 • 2 minutes to read • Edit Online

Allocates and optionally initializes a word (2 bytes) of storage for each *initializer*. Can also be used as a type specifier anywhere a type is legal. DW is a synonym of **WORD**.

Syntax

> □*name*□ **WORD** *initializer* □, *initializer* ...□

See also

Directives reference
SWORD
DW
MASM BNF Grammar

.XALL

12/20/2019 • 2 minutes to read • Edit Online

See .LISTMACRO.

Syntax

```
.XALL
```

See also

Directives reference
MASM BNF Grammar

.XCREF

12/20/2019 • 2 minutes to read • Edit Online

See .NOCREF.

Syntax

```
.XCREF
```

See also

Directives reference
MASM BNF Grammar

.XLIST

12/20/2019 • 2 minutes to read • Edit Online

See .NOLIST.

Syntax

```
.XLIST
```

See also

Directives reference
MASM BNF Grammar

.XMM (32-bit MASM)

12/20/2019 • 2 minutes to read • Edit Online

Enables assembly of Internet Streaming SIMD Extension instructions. (32-bit MASM only.)

Syntax

```
.XMM
```

See also

Directives reference
MASM BNF Grammar

XMMWORD

12/20/2019 • 2 minutes to read • Edit Online

Used for 128-bit multimedia operands with MMX and SSE (XMM) instructions.

Syntax

```
XMMWORD
```

Remarks

XMMWORD is intended to represent the same type as __m128.

Example

```
movdqa   xmm0, xmmword ptr [ebx]
```

See Also

MASM BNF Grammar

YMMWORD

12/20/2019 • 2 minutes to read • Edit Online

Used for 256-bit multimedia operands with Intel Advanced Vector Extensions (AVX) instructions.

Syntax

```
YMMWORD
```

Remarks

YMMWORD is intended to represent the same type as `__m256` for the AVX intrinsics.

See also

Directives reference
MASM BNF Grammar

Symbols reference

9/21/2020 • 2 minutes to read • Edit Online

Date and time information

`@Date`

`@Time`

Environment information

`@Cpu`

`@Environ`

`@Interface`

`@Version`

File information

`@FileCur`

`@FileName`

`@Line`

Macro functions

`@CatStr`

`@InStr`

`@SizeStr`

`@SubStr`

Miscellaneous

`$`
`?`

`@@:`

`@B`

`@F`

Segment information

`@code`
`@CodeSize`
`@CurSeg`

`@data`
`@DataSize`
`@fardata`

`@fardata?`
`@Model`

`@stack`
`@WordSize`

See also

Microsoft Macro Assembler reference
MASM BNF Grammar

$

12/20/2019 • 2 minutes to read • Edit Online

The current value of the location counter.

Syntax

```
$
```

See also

Symbols Reference
MASM BNF Grammar

?

12/20/2019 • 2 minutes to read • Edit Online

In data declarations, a value that the assembler allocates but does not initialize.

Syntax

```
?
```

See also

Symbols reference
MASM BNF Grammar

@@:

12/20/2019 • 2 minutes to read • Edit Online

Defines a code label recognizable only between *label1* and *label2*, where *label1* is either start of code or the previous `@@:` label, and *label2* is either end of code or the next `@@:` label. See @B and @F.

Syntax

```
@@:
```

See also

Symbols reference
MASM BNF Grammar

@B

12/20/2019 • 2 minutes to read • Edit Online

The location of the previous @@: label.

Syntax

```
@B
```

See also

Symbols reference
MASM BNF Grammar

@CatStr

12/20/2019 • 2 minutes to read • Edit Online

Macro function that concatenates one or more strings. Returns a string.

Syntax

> **@CatStr(** *string1* □**,** *string2*...□ **)**

See also

Symbols reference
MASM BNF Grammar

@code

12/20/2019 • 2 minutes to read • Edit Online

The name of the code segment (text macro).

Syntax

```
@code
```

See also

Symbols reference
MASM BNF Grammar

@CodeSize

12/20/2019 • 2 minutes to read • Edit Online

0 for **TINY**, **SMALL**, **COMPACT**, and **FLAT** models, and 1 for **MEDIUM**, **LARGE**, and **HUGE** models (numeric equate).

Syntax

```
@CodeSize
```

See also

Symbols reference
MASM BNF Grammar

@Cpu

12/20/2019 • 2 minutes to read • Edit Online

A bit mask specifying the processor mode (numeric equate).

Syntax

```
@Cpu
```

See also

Symbols reference
MASM BNF Grammar

@CurSeg

12/20/2019 • 2 minutes to read • Edit Online

The name of the current segment (text macro).

Syntax

```
@CurSeg
```

See also

Symbols reference
MASM BNF Grammar

@data

12/20/2019 • 2 minutes to read • Edit Online

The name of the default data group. Evaluates to DGROUP for all models except **FLAT**. Evaluates to **FLAT** under the **FLAT** memory model (text macro).

Syntax

```
@data
```

See also

Symbols reference
MASM BNF Grammar

@DataSize

12/20/2019 • 2 minutes to read • Edit Online

0 for **TINY**, **SMALL**, **MEDIUM**, and **FLAT** models, 1 for **COMPACT** and **LARGE** models, and 2 for **HUGE** model (numeric equate).

Syntax

```
@DataSize
```

See also

Symbols reference
MASM BNF Grammar

@Date

12/20/2019 • 2 minutes to read • Edit Online

The system date in the format mm/dd/yy (text macro).

Syntax

```
@Date
```

See also

Symbols reference
MASM BNF Grammar

@Environ

12/20/2019 • 2 minutes to read • Edit Online

Value of environment variable *envvar* (macro function).

Syntax

```
@Environ( envvar )
```

See also

Symbols reference
MASM BNF Grammar

@F

12/20/2019 • 2 minutes to read • Edit Online

The location of the next @@: label.

Syntax

```
@F
```

See also

Symbols reference
MASM BNF Grammar

@fardata

12/20/2019 • 2 minutes to read • Edit Online

The name of the segment defined by the .FARDATA directive (text macro).

Syntax

```
@fardata
```

See also

Symbols reference
MASM BNF Grammar

@fardata?

12/20/2019 • 2 minutes to read • Edit Online

The name of the segment defined by the .FARDATA? directive (text macro).

Syntax

```
@fardata?
```

See also

Symbols reference
MASM BNF Grammar

@FileCur

12/20/2019 • 2 minutes to read • Edit Online

The name of the current file (text macro).

Syntax

```
@FileCur
```

See also

Symbols reference
MASM BNF Grammar

@FileName

12/20/2019 • 2 minutes to read • Edit Online

The base name of the main file being assembled (text macro).

Syntax

```
@FileName
```

See also

Symbols reference
MASM BNF Grammar

@InStr

12/20/2019 • 2 minutes to read • Edit Online

Macro function that finds the first occurrence of *string2* in *string1*, beginning at *position* within *string1*. If *position* does not appear, search begins at start of *string1*. Returns a position integer or 0 if *string2* is not found.

Syntax

```
@InStr( position, string1, string2 )
```

See also

Symbols reference
MASM BNF Grammar

@Interface

12/20/2019 • 2 minutes to read • Edit Online

Information about the language parameters (numeric equate).

Syntax

```
@Interface
```

See also

Symbols reference
MASM BNF Grammar

@Line

12/20/2019 • 2 minutes to read • Edit Online

The source line number in the current file (numeric equate).

Syntax

```
@Line
```

See also

Symbols reference
MASM BNF Grammar

@Model

12/20/2019 • 2 minutes to read • Edit Online

1 for **TINY** model, 2 for **SMALL** model, 3 for **COMPACT** model, 4 for **MEDIUM** model, 5 for **LARGE** model, 6 for **HUGE** model, and 7 for **FLAT** model (numeric equate).

Syntax

```
@Model
```

See also

Symbols reference
MASM BNF Grammar

@SizeStr

12/20/2019 • 2 minutes to read • Edit Online

A macro function that returns the length of the given string. Returns an integer.

Syntax

```
@SizeStr( string )
```

See also

Symbols reference
MASM BNF Grammar

@stack

12/20/2019 • 2 minutes to read • Edit Online

DGROUP for near stacks or STACK for far stacks (text macro).

Syntax

```
@stack
```

See also

Symbols reference
MASM BNF Grammar

@SubStr

12/20/2019 • 2 minutes to read • Edit Online

A macro function that returns a substring starting at *position*.

Syntax

> **@SubStr(** *string*, *position* □, *length*□ **)**

See also

Symbols reference
MASM BNF Grammar

@Time

12/20/2019 • 2 minutes to read • Edit Online

The system time in 24-hour hh:mm:ss format (text macro).

Syntax

```
@Time
```

See also

Symbols reference
MASM BNF Grammar

@Version

12/20/2019 • 2 minutes to read • Edit Online

The major and minor version of MASM reported at the command line, as a single number (text macro). For example, MASM version 14.23.28107.0 produces "1423".

Syntax

```
@Version
```

See also

Symbols reference
MASM BNF Grammar

@WordSize

12/20/2019 • 2 minutes to read • Edit Online

Two for a 16-bit segment or four for a 32-bit segment (numeric equate).

Syntax

```
@WordSize
```

See also

Symbols reference
MASM BNF Grammar

Processor manufacturer programming manuals

9/21/2020 • 2 minutes to read • Edit Online

This article provides links to websites that may contain programming info about processors that aren't made, sold, or supported by Microsoft. Microsoft doesn't control the websites or their content.

Processor manufacturer websites

- AMD Developer Guides, Manuals & ISA Documents
- ARM Architecture Reference Manual
- Intel 64 and IA-32 Architectures Software Developer Manuals
- Introduction to Intel Advanced Vector Extensions

Remarks

Visual Studio and the Microsoft Macro Assembler don't support all processors.

See also

Microsoft Macro Assembler reference
MASM BNF Grammar

MASM Operators reference

9/21/2020 • 2 minutes to read • Edit Online

Arithmetic

`*` (multiply)
`+` (add)
`-` (subtract or negate)

`.` (field)
`/` (divide)

`[]` (index)
`MOD` (remainder)

Control Flow

`!` (runtime logical not)
`!=` (runtime not equal)
`||` (runtime logical or)
`&&` (runtime logical and)
`<` (runtime less than)

`<=` (runtime less or equal)
`==` (runtime equal)
`>` (runtime greater than)
`>=` (runtime greater or equal)
`&` (runtime bitwise and)

`CARRY?` (runtime carry test)
`OVERFLOW?` (runtime overflow test)
`PARITY?` (runtime parity test)
`SIGN?` (runtime sign test)
`ZERO?` (runtime zero test)

Logical and Shift

`AND` (bitwise and)
`NOT` (bitwise not)

`OR` (bitwise or)
`SHL` (shift bits left)

`SHR` (shift bits right)
`XOR` (bitwise exclusive or)

Macro

`!` (character literal)
`%` (treat as text)

`;;` (treat as comment)
`< >` (treat as one literal)

`& &` (substitute parameter value)

Miscellaneous

`' '` (treat as string)
`" "` (treat as string)
`:` (local label definition)

`::` (register segment and offset)
`::` (global label definition)

`;` (treat as comment)
`DUP` (repeat declaration)

Record

`MASK` (get record or field bitmask)

`WIDTH` (get record or field width)

Relational

`EQ` (equal)
`GE` (greater or equal)

`GT` (greater than)
`LE` (less or equal)

`LT` (less than)
`NE` (not equal)

Segment

`:` (segment override)
`::` (register segment and offset)
`IMAGEREL` (image relative offset)

`LROFFSET` (loader resolved offset)
`OFFSET` (segment relative offset)

`SECTIONREL` (section relative offset)
`SEG` (get segment)

Type

`HIGH` (high 8 bits of lowest 16 bits)
`HIGH32` (high 32 bits of 64 bits)
`HIGHWORD` (high 16 bits of lowest 32 bits)
`LENGTH` (number of elements in array)
`LENGTHOF` (number of elements in array)
`LOW` (low 8 bits)

`LOW32` (low 32 bits)

`LOWWORD` (low 16 bits)
`OPATTR` (get argument type info)
`PTR` (pointer to or as type)
`SHORT` (mark short label type)

`SIZE` (size of type or variable)
`SIZEOF` (size of type or variable)
`THIS` (current location)
`TYPE` (get expression type)
`.TYPE` (get argument type info)

See also

Microsoft Macro Assembler Reference
MASM BNF Grammar

operator +

12/20/2019 • 2 minutes to read • Edit Online

The first operator returns *expression1* plus *expression2*.

Syntax

> *expression1* + *expression2*

See also

Operators Reference
MASM BNF Grammar

operator -

12/20/2019 • 2 minutes to read • Edit Online

Returns *expression1* minus *expression2*. The second operator reverses the sign of *expression*.

Syntax

> *expression1* - *expression2*
>
> -*expression*

See also

Operators reference
MASM BNF Grammar

operator *

12/20/2019 • 2 minutes to read • Edit Online

Returns *expression1* times *expression2*.

Syntax

> *expression1* * *expression2*

See also

Operators reference
MASM BNF Grammar

operator /

12/20/2019 • 2 minutes to read • Edit Online

Returns *expression1* divided by *expression2*.

Syntax

> *expression1* **/** *expression2*

See also

Operators reference
MASM BNF Grammar

operator []

12/20/2019 • 2 minutes to read • Edit Online

Returns *expression1* plus [*expression2*].

Syntax

> *expression1* [expression2]

See also

Operators Reference
MASM BNF Grammar

operator :

12/20/2019 • 2 minutes to read • Edit Online

Overrides the default segment of *expression* with *segment*. The *segment* can be a segment register, group name, segment name, or segment expression. The *expression* must be a constant.

Syntax

> *segment*: *expression*

See also

Operators reference
MASM BNF Grammar

operator .

12/20/2019 • 2 minutes to read • Edit Online

The first operator returns *expression* plus the offset of *field* within its structure or union. The second operator returns value at the location pointed to by *register* plus the offset of *field* within its structure or union.

Syntax

> *expression*.*field*□.*field* ...□
>
> [*register*].*field*□.*field* ...□

See also

Operators reference
MASM BNF Grammar

operator < >

12/20/2019 • 2 minutes to read • Edit Online

Treats *text* as a single literal element.

Syntax

```
< text>
```

See also

Operators reference
MASM BNF Grammar

operator " "

12/20/2019 • 2 minutes to read • Edit Online

Treats "*text*" as a string.

Syntax

> " *text*"

See also

Operators reference
MASM BNF Grammar

operator ' '

12/20/2019 • 2 minutes to read • Edit Online

Treats '*text*' as a string.

Syntax

```
'text'
```

See also

Operators reference

MASM BNF Grammar

operator ! (MASM)

12/20/2019 • 2 minutes to read • Edit Online

Treats *character* as a literal character rather than as an operator or symbol.

Syntax

> ! *character*

See also

Operators reference
MASM BNF Grammar

operator ;

12/20/2019 • 2 minutes to read • Edit Online

Treats *text* as a comment.

Syntax

```
; text
```

See also

Operators reference
MASM BNF Grammar

operator ;;

12/20/2019 • 2 minutes to read • Edit Online

Treats *text* as a comment in a macro that appears only in the macro definition. The listing does not show *text* where the macro is expanded.

Syntax

```
;; text
```

See also

Operators reference
MASM BNF Grammar

operator %

12/20/2019 • 2 minutes to read • Edit Online

Treats the value of *expression* in a macro argument as text.

Syntax

```
% expression
```

See also

Operators reference
MASM BNF Grammar

Substitution operator (MASM)

12/20/2019 • 2 minutes to read • Edit Online

Replaces *parameter* with its corresponding argument value.

Syntax

> &*parameter*&

See also

Operators reference
MASM BNF Grammar

operator ABS

12/20/2019 • 2 minutes to read • Edit Online

See the EXTERNDEF directive.

Syntax

```
ABS
```

See also

Operators reference
MASM BNF Grammar

operator ADDR

12/20/2019 • 2 minutes to read • Edit Online

See the INVOKE directive.

Syntax

```
ADDR
```

See also

Operators Reference
MASM BNF Grammar

operator AND

3/16/2020 • 2 minutes to read • Edit Online

Returns the result of a bitwise AND operation for *expression1* and *expression2*.

Syntax

> *expression1* **AND** *expression2*

See also

Operators Reference
MASM BNF Grammar

operator DUP

12/20/2019 • 2 minutes to read • Edit Online

Specifies *count* number of declarations of *initialvalue*.

Syntax

> *count* **DUP** (*initialvalue* , *initialvalue* ...)

See also

Operators reference
MASM BNF Grammar

operator EQ

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* equals *expression2*, or returns false (0) if it does not.

Syntax

> *expression1* **EQ** *expression2*

See also

Operators reference
MASM BNF Grammar

operator GE

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* is greater than or equal to *expression2*, or returns false (0) if it is not.

Syntax

> *expression1* **GE** *expression2*

See also

Operators reference
MASM BNF Grammar

operator GT

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* is greater than *expression2*, or returns false (0) if it is not.

Syntax

> *expression1* **GT** *expression2*

See also

Operators reference
MASM BNF Grammar

operator HIGH

12/20/2019 • 2 minutes to read • Edit Online

Returns the high 8 bits of the low 16 bits of *expression*. MASM expressions are 64-bit values.

Syntax

> **HIGH** *expression*

See also

Operators reference
MASM BNF Grammar

operator HIGH32

12/20/2019 • 2 minutes to read • Edit Online

Returns the high 32 bits of *expression*. MASM expressions are 64-bit values.

Syntax

> **HIGH32** *expression*

See also

Operators reference
MASM BNF Grammar

operator HIGHWORD

12/20/2019 • 2 minutes to read • Edit Online

Returns the high 16 bits of the low 32 bits of *expression*. MASM expressions are 64-bit values.

Syntax

> HIGHWORD *expression*

See also

Operators reference
MASM BNF Grammar

operator IMAGEREL

12/20/2019 • 2 minutes to read • Edit Online

Returns the image relative offset of *expression*.

Syntax

```
IMAGEREL expression
```

Remarks

The resulting value is often referred to as an RVA or Relative Virtual Address.

IMAGEREL is available only with COFF object emission.

See also

Operators reference
MASM BNF Grammar

operator LE

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* is less than or equal to *expression2*, or returns false (0) if it is not.

Syntax

> *expression1* **LE** *expression2*

See also

Operators reference
MASM BNF Grammar

operator LENGTH

3/16/2020 • 2 minutes to read • Edit Online

Returns the number of data items in *variable* created by the first initializer.

Syntax

```
LENGTH variable
```

See also

Operators reference
MASM BNF Grammar

operator LENGTHOF

12/20/2019 • 2 minutes to read • Edit Online

Returns the number of data objects in *variable*.

Syntax

```
LENGTHOF variable
```

See also

Operators reference
MASM BNF Grammar

operator LOW

12/20/2019 • 2 minutes to read • Edit Online

Returns the low 8 bits of *expression*. MASM expressions are 64-bit values.

Syntax

> **LOW** *expression*

See also

Operators reference
MASM BNF Grammar

operator LOW32

12/20/2019 • 2 minutes to read • Edit Online

Returns the low 32 bits of *expression*. MASM expressions are 64-bit values.

Syntax

```
LOW32 expression
```

See also

Operators reference
MASM BNF Grammar

operator LOWWORD

12/20/2019 • 2 minutes to read • Edit Online

Returns the low 16 bits of *expression*. MASM expressions are 64-bit values.

Syntax

> **LOWWORD** *expression*

See also

Operators reference
MASM BNF Grammar

operator LROFFSET

12/20/2019 • 2 minutes to read • Edit Online

Returns the offset of *expression*. Same as **OFFSET**, but it generates a loader resolved offset, which allows Windows to relocate code segments.

Syntax

> **LROFFSET** *expression*

See also

Operators reference
MASM BNF Grammar

operator LT

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* is less than *expression2*, or returns false (0) if it is not.

Syntax

> *expression1* **LT** *expression2*

See also

Operators reference
MASM BNF Grammar

operator MASK

12/20/2019 • 2 minutes to read • Edit Online

Returns a bit mask in which the bits in *recordfieldname* or *record* are set and all other bits are cleared.

Syntax

> **MASK** {*recordfieldname* | *record*}

See also

Operators reference
MASM BNF Grammar

operator MOD

12/20/2019 • 2 minutes to read • Edit Online

Returns the integer value of the remainder (modulo) when dividing *expression1* by *expression2*.

Syntax

> *expression1* **MOD** *expression2*

See also

Operators reference
MASM BNF Grammar

operator NE

12/20/2019 • 2 minutes to read • Edit Online

Returns true (-1) if *expression1* does not equal *expression2*, or returns false (0) if it does.

Syntax

> *expression1* **NE** *expression2*

See also

Operators reference
MASM BNF Grammar

operator NOT

12/20/2019 • 2 minutes to read • Edit Online

Returns *expression* with all bits reversed.

Syntax

```
NOT expression
```

See also

Operators reference
MASM BNF Grammar

operator OFFSET

12/20/2019 • 2 minutes to read • Edit Online

Returns the offset into the relevant segment of *expression*.

Syntax

> OFFSET *expression*

See also

Operators reference
MASM BNF Grammar

operator OPATTR

12/20/2019 • 2 minutes to read • Edit Online

Returns a word defining the mode and scope of *expression*. The low byte is identical to the byte returned by .TYPE. The high byte contains additional information.

Syntax

```
OPATTR expression
```

See also

Operators reference
MASM BNF Grammar

operator OR

3/16/2020 • 2 minutes to read • Edit Online

Returns the result of a bitwise OR operation for *expression1* and *expression2*.

Syntax

> *expression1* **OR** *expression2*

See also

Operators reference
MASM BNF Grammar

operator PTR

12/20/2019 • 2 minutes to read • Edit Online

The first operator forces the *expression* to be treated as having the specified *type*. The second operator specifies a pointer to *type*.

Syntax

> *type* **PTR** *expression*
>
> [*distance*] **PTR** *type*

See also

Operators reference
MASM BNF Grammar

operator SEG

12/20/2019 • 2 minutes to read • Edit Online

Returns the segment of *expression*.

Syntax

```
SEG expression
```

See also

Operators reference
MASM BNF Grammar

operator SHL

12/20/2019 • 2 minutes to read • Edit Online

Returns the result of shifting the bits of *expression* left *count* number of bits.

Syntax

> *expression* **SHL** *count*

See also

Operators reference
MASM BNF Grammar

operator .TYPE

12/20/2019 • 2 minutes to read • Edit Online

See OPATTR.

Syntax

```
.TYPE expression
```

See also

Operators reference
MASM BNF Grammar

operator SECTIONREL

12/20/2019 • 2 minutes to read • Edit Online

Returns the section relative offset of expression relative to the section containing the target in the final executable.

Syntax

> **SECTIONREL** *expression*

Remarks

SECTIONREL is available only with COFF object emission.

See also

Operators reference
MASM BNF Grammar

operator SHORT

12/20/2019 • 2 minutes to read • Edit Online

Sets the type of *label* to short. All jumps to *label* must be short (within the range -128 to +127 bytes from the jump instruction to *label*).

Syntax

```
SHORT label
```

See also

Operators reference
MASM BNF Grammar

operator SHR

12/20/2019 • 2 minutes to read • Edit Online

Returns the result of shifting the bits of *expression* right *count* number of bits.

Syntax

> *expression* **SHR** *count*

See also

Operators reference
MASM BNF Grammar

operator SIZE

12/20/2019 • 2 minutes to read • Edit Online

Returns the number of bytes in *variable* allocated by the first initializer.

Syntax

```
SIZE variable
```

See also

Operators reference
MASM BNF Grammar

operator SIZEOF

12/20/2019 • 2 minutes to read • Edit Online

Returns the number of bytes in *variable* or *type*.

Syntax

> SIZEOF {*variable* | *type*}

See also

Operators reference
MASM BNF Grammar

operator THIS

12/20/2019 • 2 minutes to read • Edit Online

Returns an operand of specified *type* whose offset and segment values are equal to the current location counter value.

Syntax

```
THIS type
```

See also

Operators reference
MASM BNF Grammar

operator TYPE

3/16/2020 • 2 minutes to read • Edit Online

Returns the type of *expression*.

Syntax

```
TYPE expression
```

See also

Operators reference
MASM BNF Grammar

operator WIDTH

12/20/2019 • 2 minutes to read • Edit Online

Returns the width in bits of the current *recordfieldname* or *record*.

Syntax

> **WIDTH** {*recordfieldname* | *record*}

See also

Operators reference
MASM BNF Grammar

operator XOR

12/20/2019 • 2 minutes to read • Edit Online

Returns the result of a bitwise **XOR** operation for *expression1* and *expression2*.

Syntax

> *expression1* **XOR** *expression2*

See also

Operators reference
MASM BNF Grammar

operator == (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Is equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* == *expression2*

See also

Operators reference
MASM BNF Grammar

operator != (MASM)

12/20/2019 • 2 minutes to read • Edit Online

Is not equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* **!=** *expression2*

See also

Operators reference
MASM BNF Grammar

operator > (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Is greater than. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
expression1 > expression2
```

See also

Operators reference
MASM BNF Grammar

operator >= (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Is greater than or equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* >= *expression2*

See also

Operators reference
MASM BNF Grammar

operator < (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Is less than. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
expression1 < expression2
```

See also

Operators reference
MASM BNF Grammar

operator <= (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Is less than or equal to. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* <= *expression2*

See also

Operators reference
MASM BNF Grammar

operator ||

12/20/2019 • 2 minutes to read • Edit Online

Logical OR. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
expression1 || expression2
```

See also

Operators reference
MASM BNF Grammar

operator && (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Logical **AND**. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* **&&** *expression2*

See also

Operators reference
MASM BNF Grammar

operator &

12/20/2019 • 2 minutes to read • Edit Online

Bitwise **AND**. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> *expression1* & *expression2*

See also

Operators Reference
MASM BNF Grammar

operator ! (MASM Run Time)

12/20/2019 • 2 minutes to read • Edit Online

Logical negation. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

> ! *expression*

See also

Operators reference
MASM BNF Grammar

operator CARRY?

12/20/2019 • 2 minutes to read • Edit Online

Status of carry flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
CARRY?
```

See also

Operators reference
MASM BNF Grammar

operator OVERFLOW?

12/20/2019 • 2 minutes to read • Edit Online

Status of overflow flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
OVERFLOW?
```

See also

Operators reference
MASM BNF Grammar

operator PARITY?

12/20/2019 • 2 minutes to read • Edit Online

Status of parity flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
PARITY?
```

See also

Operators reference
MASM BNF Grammar

operator SIGN?

12/20/2019 • 2 minutes to read • Edit Online

Status of sign flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
SIGN?
```

See also

Operators reference
MASM BNF Grammar

operator ZERO?

12/20/2019 • 2 minutes to read • Edit Online

Status of zero flag. Used only within .IF, .WHILE, or .REPEAT blocks and evaluated at run time, not at assembly time.

Syntax

```
ZERO?
```

See also

Operators reference
MASM BNF Grammar

ML Error Messages

12/20/2019 • 2 minutes to read • Edit Online

The error messages generated by MASM components fall into three categories:

- **Fatal errors.** These indicate a severe problem that prevents the utility from completing its normal process.
- **Nonfatal errors.** The utility may complete its process. If it does, its result is not likely to be the one you want.
- **Warnings.** These messages indicate conditions that may prevent you from getting the results you want.

All error messages take the following form:

> *Utility*: *Filename* (*Line*) : {*Error_type*} (*Code*): *Message_text*

where:

Utility
The program that sent the error message.

Filename
The file that contains the error-generating condition.

Line
The approximate line where the error condition exists.

Error_type
Fatal Error, Error, or Warning.

Code
The unique 5- or 6-digit error code.

Message_text
A short and general description of the error condition.

See also

Microsoft Macro Assembler reference

ML Fatal Error A1000

12/20/2019 • 2 minutes to read • Edit Online

cannot open file: filename

The assembler was unable to open a source, include, or output file.

One of the following may be a cause:

- The file does not exist.
- The file is in use by another process.
- The filename is not valid.
- A read-only file with the output filename already exists.
- The current drive is full.
- The current directory is the root and is full.
- The device cannot be written to.
- The drive is not ready.

See also

ML Error Messages

ML Fatal Error A1005

12/20/2019 • 2 minutes to read • Edit Online

assembler limit : macro parameter name table full

Too many parameters, locals, or macro labels were defined for a macro. There was no more room in the macro name table.

Define shorter or fewer names, or remove unnecessary macros.

See also

ML Error Messages

ML Fatal Error A1007

12/20/2019 • 2 minutes to read • Edit Online

nesting level too deep

The assembler reached its nesting limit. The limit is 20 levels except where noted otherwise.

One of the following was nested too deeply:

- A high-level directive such as .IF, .REPEAT, or .WHILE.
- A structure definition.
- A conditional-assembly directive.
- A procedure definition.
- A PUSHCONTEXT directive (the limit is 10).
- A segment definition.
- An include file.
- A macro.

See also

ML Error Messages

ML Fatal Error A1008

12/20/2019 • 2 minutes to read • Edit Online

unmatched macro nesting

Either a macro was not terminated before the end of the file or the terminating directive ENDM was found outside of a macro block.

One cause of this error is omission of the dot before .REPEAT or .WHILE.

See also

ML Error Messages

ML Fatal Error A1009

12/20/2019 • 2 minutes to read • Edit Online

line too long

A line in a source file exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line-continuation character (\ ), then the resulting logical line is still limited to 512 characters.

See also

ML Error Messages

ML Fatal Error A1010

12/20/2019 • 2 minutes to read • Edit Online

unmatched block nesting :

A block beginning did not have a matching end, or a block end did not have a matching beginning. One of the following may be involved:

- A high-level directive such as .IF, .REPEAT, or .WHILE.
- A conditional-assembly directive such as IF, REPEAT, or **WHILE**.
- A structure or union definition.
- A procedure definition.
- A segment definition.
- A POPCONTEXT directive.
- A conditional-assembly directive, such as an ELSE, ELSEIF, or **ENDIF** without a matching IF.

See also

ML Error Messages

ML Fatal Error A1011

12/20/2019 • 2 minutes to read • Edit Online

directive must be in control block

The assembler found a high-level directive where one was not expected. One of the following directives was found:

- .ELSE without .IF
- .ENDIF without .IF
- .ENDW without .WHILE
- .UNTILCXZ without .REPEAT
- .CONTINUE without .WHILE or .REPEAT
- .BREAK without .WHILE or .REPEAT
- .ELSE following `.ELSE`

See also

ML Error Messages

ML Fatal Error A1016

3/19/2020 • 2 minutes to read • Edit Online

Internal Assembler Error

The MASM driver, called ML.exe, generated a system error.

Note the circumstances of the error and notify Microsoft Corporation. Product Support Services is available at https://support.microsoft.com/.

See also

ML Error Messages

ML Fatal Error A1017

12/20/2019 • 2 minutes to read • Edit Online

missing source filename

ML could not find a file to assemble or pass to the linker.

This error is generated when you give ML command-line options without specifying a filename to act upon. To assemble files that do not have an .asm extension, use the **/Ta** command-line option.

This error can also be generated by invoking ML with no parameters if the ML environment variable contains command-line options.

See also

ML Error Messages

ML Nonfatal Error A2004

12/20/2019 • 2 minutes to read • Edit Online

symbol type conflict : identifier

The EXTERNDEF or LABEL directive was used on a variable, symbol, data structure, or label that was defined in the same module but with a different type.

See also

ML Error Messages

ML Nonfatal Error A2006

12/20/2019 • 2 minutes to read • Edit Online

undefined symbol : identifier

An attempt was made to use a symbol that was not defined.

One of the following may have occurred:

- A symbol was not defined.
- A field was not a member of the specified structure.
- A symbol was defined in an include file that was not included.
- An external symbol was used without an EXTERN or EXTERNDEF directive.
- A symbol name was misspelled.
- A local code label was referenced outside of its scope.

See also

ML Error Messages

ML Nonfatal Error A2008

12/20/2019 • 2 minutes to read • Edit Online

syntax error :

A token at the current location caused a syntax error.

One of the following may have occurred:

- A dot prefix was added to or omitted from a directive.
- A reserved word (such as **C** or **SIZE**) was used as an identifier.
- An instruction was used that was not available with the current processor or coprocessor selection.
- A comparison run-time operator (such as `==`) was used in a conditional assembly statement instead of a relational operator (such as EQ).
- An instruction or directive was given too few operands.
- An obsolete directive was used.

See also

ML Error Messages

ML Nonfatal Error A2010

12/20/2019 • 2 minutes to read • Edit Online

invalid type expression

The operand to THIS or PTR was not a valid type expression.

See also

ML Error Messages

ML Nonfatal Error A2019

12/20/2019 • 2 minutes to read • Edit Online

operand must be RECORD type or field

The operand following the WIDTH or MASK operator was not valid.

The WIDTH operator takes an operand that is the name of a field or a record. The MASK operator takes an operand that is the name of a field or a record type.

See also

ML Error Messages

ML Nonfatal Error A2022

12/20/2019 • 2 minutes to read • Edit Online

instruction operands must be the same size

The operands to an instruction did not have the same size.

See also

ML Error Messages

ML Nonfatal Error A2031

12/20/2019 • 2 minutes to read • Edit Online

must be index or base register

An attempt was made to use a register that was not a base or index register in a memory expression.

For example, the following expressions cause this error:

```
[ax]
[bl]
```

See also

ML Error Messages

ML Nonfatal Error A2034

12/20/2019 • 2 minutes to read • Edit Online

must be in segment block

One of the following was found outside of a segment block:

- An instruction
- A label definition
- A THIS operator
- A $ operator
- A procedure definition
- An ALIGN directive
- An ORG directive

See also

ML Error Messages

ML Nonfatal Error A2037

12/20/2019 • 2 minutes to read • Edit Online

statement not allowed inside structure definition

A structure definition contained an invalid statement.

A structure cannot contain instructions, labels, procedures, control-flow directives, .STARTUP, or .EXIT.

See also

ML Error Messages

ML Nonfatal Error A2038

12/20/2019 • 2 minutes to read • Edit Online

missing operand for macro operator

The assembler found the end of a macro's parameter list immediately after the ! or % operator.

See also

ML Error Messages

ML Nonfatal Error A2039

12/20/2019 • 2 minutes to read • Edit Online

line too long

A source-file line exceeded the limit of 512 characters.

If multiple physical lines are concatenated with the line-continuation character (\ ), then the resulting logical line is still limited to 512 characters.

See also

ML Error Messages

ML Nonfatal Error A2044

12/20/2019 • 2 minutes to read • Edit Online

invalid character in file

The source file contained a character outside a comment, string, or literal that was not recognized as an operator or other legal character.

See also

ML Error Messages

ML Nonfatal Error A2047

12/20/2019 • 2 minutes to read • Edit Online

empty (null) string

A string consisted of a delimiting pair of quotation marks and no characters within.

For a string to be valid, it must contain 1-255 characters.

See also

ML Error Messages

ML Nonfatal Error A2050

12/20/2019 • 2 minutes to read • Edit Online

real or BCD number not allowed

A floating-point (real) number or binary coded decimal (BCD) constant was used other than as a data initializer.

One of the following occurred:

- A real number or a BCD was used in an expression.
- A real number was used to initialize a directive other than DWORD, QWORD, or TBYTE.
- A BCD was used to initialize a directive other than `TBYTE`.

See also

ML Error Messages

ML Nonfatal Error A2054

12/20/2019 • 2 minutes to read • Edit Online

forced error : value not equal to 0

The conditional-error directive .ERRNZ was used to generate this error.

See also

ML Error Messages

ML Nonfatal Error A2055

12/20/2019 • 2 minutes to read • Edit Online

forced error : symbol not defined

The conditional-error directive .ERRNDEF was used to generate this error.

See also

ML Error Messages

ML Nonfatal Error A2057

12/20/2019 • 2 minutes to read • Edit Online

forced error : string blank

The conditional-error directive .ERRB was used to generate this error.

See also

ML Error Messages

ML Nonfatal Error A2059

12/20/2019 • 2 minutes to read • Edit Online

forced error : strings equal

The conditional-error directive .ERRIDN or **.ERRIDNI** was used to generate this error.

See also

ML Error Messages

ML Nonfatal Error A2060

12/20/2019 • 2 minutes to read • Edit Online

forced error : strings not equal

The conditional-error directive .ERRDIF or **.ERRDIFI** was used to generate this error.

See also

ML Error Messages

ML Nonfatal Error A2063

12/20/2019 • 2 minutes to read • Edit Online

can ALIGN only to power of 2 : expression

The expression specified with the ALIGN directive was invalid.

The **ALIGN** expression must be a power of 2 between 2 and 256, and must be less than or equal to the alignment of the current segment, structure, or union.

See also

ML Error Messages

ML Nonfatal Error A2064

12/20/2019 • 2 minutes to read • Edit Online

structure alignment must be 1, 2, 4, 8, or 16

The alignment specified in a structure definition was invalid.

See also

ML Error Messages

ML Nonfatal Error A2065

12/20/2019 • 2 minutes to read • Edit Online

expected : token

The assembler expected the given token.

See also

ML Error Messages

ML Nonfatal Error A2066

12/20/2019 • 2 minutes to read • Edit Online

incompatible CPU mode and segment size

An attempt was made to open a segment with a **USE16**, **USE32**, or **FLAT** attribute that was not compatible with the specified CPU, or to change to a 16-bit CPU while in a 32-bit segment.

The **USE32** and **FLAT** attributes must be preceded by the .386 or greater processor directive.

See also

ML Error Messages

ML Nonfatal Error A2069

12/20/2019 • 2 minutes to read • Edit Online

no operands allowed for this instruction

One or more operands were specified with an instruction that takes no operands.

See also

ML Error Messages

ML Nonfatal Error A2070

12/20/2019 • 2 minutes to read • Edit Online

invalid instruction operands

One or more operands were not valid for the instruction with which they were specified.

See also

ML Error Messages

ML Nonfatal Error A2074

12/20/2019 • 2 minutes to read • Edit Online

cannot access label through segment registers

An attempt was made to access a label through a segment register that was not assumed to its segment or group.

See also

ML Error Messages

ML Nonfatal Error A2078

12/20/2019 • 2 minutes to read • Edit Online

instruction does not allow FAR indirect addressing

A conditional jump or loop cannot take a memory operand. It must be given a relative address or label.

See also

ML Error Messages

ML Nonfatal Error A2079

12/20/2019 • 2 minutes to read • Edit Online

instruction does not allow FAR direct addressing

A conditional jump or loop cannot be to a different segment or group.

See also

ML Error Messages

ML Nonfatal Error A2083

12/20/2019 • 2 minutes to read • Edit Online

invalid scale value

A register scale was specified that was not 1, 2, 4, or 8.

See also

ML Error Messages

ML Nonfatal Error A2085

12/20/2019 • 2 minutes to read • Edit Online

instruction or register not accepted in current CPU mode

An attempt was made to use an instruction, register, or keyword that was not valid for the current processor mode.

For example, 32-bit registers require .386 or above. Control registers such as CR0 require privileged mode .386P or above. This error will also be generated for the **NEAR32**, **FAR32**, and **FLAT** keywords, which require **.386** or above.

See also

ML Error Messages

ML Nonfatal Error A2096

12/20/2019 • 2 minutes to read • Edit Online

segment, group, or segment register expected

A segment or group was expected but was not found.

One of the following occurred:

- The left operand specified with the segment override operator (:) was not a segment register (CS, DS, SS, ES, FS, or GS), group name, segment name, or segment expression.
- The ASSUME directive was given a segment register without a valid segment address, segment register, group, or the special **FLAT** group.

See also

ML Error Messages

ML Nonfatal Error A2097

12/20/2019 • 2 minutes to read • Edit Online

segment expected : identifier

The GROUP directive was given an identifier that was not a defined segment.

See also

ML Error Messages

ML Nonfatal Error A2107

12/20/2019 • 2 minutes to read • Edit Online

cannot have implicit far jump or call to near label

An attempt was made to make an implicit far jump or call to a near label in another segment.

See also

ML Error Messages

ML Nonfatal Error A2119

12/20/2019 • 2 minutes to read • Edit Online

language type must be specified

A procedure definition or prototype was not given a language type.

A language type must be declared in each procedure definition or prototype if a default language type is not specified. A default language type is set using either the .MODEL directive, **OPTION LANG**, or the ML command-line options **/Gc** or **/Gd**.

See also

ML Error Messages

ML Nonfatal Error A2133

12/20/2019 • 2 minutes to read • Edit Online

register value overwritten by INVOKE

A register was passed as an argument to a procedure, but the code generated by INVOKE to pass other arguments destroyed the contents of the register.

The AX, AL, AH, EAX, DX, DL, DH, and EDX registers may be used by the assembler to perform data conversion.

Use a different register.

See also

ML Error Messages

ML Nonfatal Error A2137

12/20/2019 • 2 minutes to read • Edit Online

too few arguments to INVOKE

The number of arguments passed using the INVOKE directive was fewer than the number of required parameters specified in the prototype for the procedure being invoked.

See also

ML Error Messages

ML Nonfatal Error A2189

12/20/2019 • 2 minutes to read • Edit Online

invalid combination with segment alignment

The alignment specified by the **ALIGN** or `EVEN` directive was greater than the current segment alignment as specified by the **SEGMENT** directive.

See also

ML Error Messages

ML Nonfatal Error A2206

12/20/2019 • 2 minutes to read • Edit Online

missing operator in expression

An expression cannot be evaluated because it is missing an operator. This error message may also be a side-effect of a preceding program error.

The following line will generate this error:

```
value1 = ( 1 + 2 ) 3
```

See also

ML Error Messages

ML Nonfatal Error A2219

12/20/2019 • 2 minutes to read • Edit Online

> Bad alignment for offset in unwind code

Remarks

The operand for .ALLOCSTACK and .SAVEREG must be a multiple of 8. The operand for .SAVEXMM128 and .SETFRAME must be a multiple of 16.

See also

ML error messages

ML Warning A4004

12/20/2019 • 2 minutes to read • Edit Online

cannot ASSUME CS

An attempt was made to assume a value for the CS register. CS is always set to the current segment or group.

See also

ML Error Messages

ML Warning A4012

12/20/2019 • 2 minutes to read • Edit Online

line number information for segment without class 'CODE'

There were instructions in a segment that did not have a class name that ends with "CODE." The assembler did not generate CodeView information for these instructions.

CodeView cannot process modules with code in segments with class names that do not end with "CODE."

See also

ML Error Messages

ML Warning A4014

12/20/2019 • 2 minutes to read • Edit Online

instructions and initialized data not supported in BSS segments

An attempt was made to define initialized data within a BSS section. A BSS section is defined as a class whose name is BSS. This includes the simplified segment `.data?`.

See also

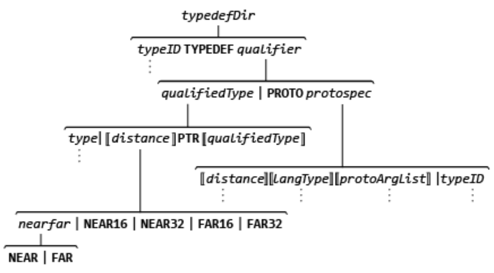
ML Error Messages

Microsoft Macro Assembler BNF Grammar

3/27/2020 • 14 minutes to read • Edit Online

This page contains a BNF description of the MASM grammar. It's provided as a supplement to the reference and isn't guaranteed to be complete. Consult the reference for full information on keywords, parameters, operations, and so on.

To illustrate the use of the BNF, the following diagram shows the definition of the TYPEDEF directive, starting with the nonterminal `typedefDir`.


typedefDir
typeID TYPEDEF qualifier
qualifiedType | PROTO protospec
type| ⟦distance⟧PTR ⟦qualifiedType⟧
⟦distance⟧⟦langType⟧⟦protoArgList⟧ |typeID
nearfar | NEAR16 | NEAR32 | FAR16 | FAR32
NEAR | FAR

The entries under each horizontal brace are terminals, such as `NEAR16`, `NEAR32`, `FAR16`, and `FAR32`. Or, they're nonterminals such as `qualifier`, `qualifiedType`, `distance`, and `protoSpec` that can be further defined. Each italicized nonterminal in the `typedefDir` definition is also an entry in the BNF. Three vertical dots indicate a branching definition for a nonterminal that, for the sake of simplicity, this figure doesn't illustrate.

The BNF grammar allows recursive definitions. For example, the grammar uses qualifiedType as a possible definition for qualifiedType, which is also a component of the definition for qualifier. The "|" symbol specifies a choice between alternate expressions, for example `endOfLine` | `comment`. Double braces specify an optional parameter, for example ⟦ `macroParmList` ⟧. The brackets don't actually appear in the source code.

MASM Nonterminals

`;;`
  `endOfLine` | `comment`

`=Dir`
  `id` = `immExpr` `;;`

`addOp`
  `+` | `-`

`aExpr`
  `term` | `aExpr` `&&` `term`

`altId`
  `id`

`arbitraryText`
  `charList`

`asmInstruction`
  `mnemonic` ⟦ `exprList` ⟧

`assumeDir`
  `ASSUME` `assumeList` `;;`
  | `ASSUME NOTHING` `;;`

`assumeList`
  `assumeRegister` | `assumeList` `,`
`assumeRegister`\

`assumeReg`
  `register` `:` `assumeVal`

`assumeRegister`
  `assumeSegReg` | `assumeReg`

`assumeSegReg`
  `segmentRegister` `:` `assumeSegVal`

`assumeSegVal`
  `frameExpr` | `NOTHING` | `ERROR`

`assumeVal`
  `qualifiedType` | `NOTHING` | `ERROR`

`bcdConst`
  ⟦ `sign` ⟧ `decNumber`

`binaryOp`
  `==` | `!=` | `>=` | `<=` | `>` | `<` | `&`

`bitDef`
  `bitFieldId` `:` `bitFieldSize` ⟦ `=`
`constExpr` ⟧

`bitDefList`

`bitDef` | `bitDefList` `,` ⟦ `;;` ⟧ `bitDef`

`bitFieldId`
`id`

`bitFieldSize`
`constExpr`

`blockStatements`
`directiveList`
| `.CONTINUE` `.IF` `cExpr` ⟧
| `.BREAK` ⟦ `.IF` `cExpr` ⟧

`bool`
`TRUE` | `FALSE`

`byteRegister`
`AL` | `AH` | `CL` | `CH` | `DL` | `DH` | `BL` |
`BH` | `R8B` | `R9B` | `R10B` | `R11B` | `R12B` |
`R13B` | `R14B` | `R15B`

`cExpr`
`aExpr` | `cExpr` `||` `aExpr`

`character`
Any character with ordinal in the range 0–255 except linefeed (10).

`charList`
`character` | `charList` `character`

`className`
`string`

`commDecl`
⟦ `nearfar` ⟧ ⟦ `langType` ⟧ `id` `:`
`commType`
⟦ `:` `constExpr` ⟧

`commDir`
`COMM`
`commList` `;;`

`comment`
; `text` `;;`

`commentDir`
`COMMENT` `delimiter`
`text`
`text` `delimiter` `text` `;;`

`commList`
`commDecl` | `commList` `,` `commDecl`

`commType`
`type` | `constExpr`

constant
  digits ⟦ *radixOverride* ⟧

constExpr
  expr

contextDir
  PUSHCONTEXT *contextItemList* *;;*
  POPCONTEXT *contextItemList* *;;*

contextItem
  ASSUMES | RADIX | LISTING | CPU | ALL

contextItemList
  contextItem | *contextItemList* *,*
contextItem

controlBlock
  whileBlock | *repeatBlock*

controlDir
  controlIf | *controlBlock*

controlElseif
  .ELSEIF *cExpr* *;;*
  directiveList
  ⟦ *controlElseif* ⟧

controlIf
  .IF *cExpr* *;;*
  directiveList
  ⟦ *controlElseif* ⟧
  ⟦ .ELSE *;;*
  [*directiveList* ⟧
  .ENDIF *;;*

coprocessor
  .8087 | .287 | .387 | .NO87

crefDir
  crefOption *;;*

crefOption
  .CREF
  | .XCREF ⟦ *idList* ⟧
  | .NOCREF ⟦ *idList* ⟧

cxzExpr
  expr
  | ! *expr*
  | *expr* == *expr*
  | *expr* != *expr*

dataDecl
  DB | DW | DD | DF | DQ | DT |
dataType | *typeId*

`dataDir`
□ `id` □ `dataItem` `;;`

`dataItem`
`dataDecl` `scalarInstList`
| `structTag` `structInstList`
| `typeId` `structInstList`
| `unionTag` `structInstList`
| `recordTag` `recordInstList`

`dataType`
`BYTE` | `SBYTE` | `WORD` | `SWORD` | `DWORD` | `SDWORD` | `FWORD` | `QWORD` | `SQWORD` | `TBYTE` | `OWORD` | `REAL4` | `REAL8` | `REAL10` | `MMWORD` | `XMMWORD` | `YMMWORD`

`decdigit`
`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

`decNumber`
`decdigit`
| `decNumber` `decdigit`

`delimiter`
Any character except `whiteSpaceCharacter`

`digits`
`decdigit`
| `digits` `decdigit`
| `digits` `hexdigit`

`directive`
`generalDir` | `segmentDef`

`directiveList`
`directive` | `directiveList` `directive`

`distance`
`nearfar` | `NEAR16` | `NEAR32` | `FAR16` | `FAR32`

`e01`
`e01` `orOp` `e02` | `e02`

`e02`
`e02` `AND` `e03` | `e03`

`e03`
`NOT` `e04` | `e04`

`e04`
`e04` `relOp` `e05` | `e05`

`e05`

```
  e05 addOp e06 | e06

e06
  e06 mulOp e07 | e06 shiftOp e07 |
e07

e07
  e07 addOp e08 | e08

e08
  HIGH e09
  | LOW e09
  | HIGHWORD e09
  | LOWWORD e09
  | e09

e09
  OFFSET e10
  | SEG e10
  | LROFFSET e10
  | TYPE e10
  | THIS e10
  | e09 PTR e10
  | e09 : e10
  | e10

e10
  e10 . e11
  | e10 □ expr □
  | e11

e11
  ( expr )
  | □ expr □
  | WIDTH id
  | MASK id
  | SIZE sizeArg
  | SIZEOF sizeArg
  | LENGTH id
  | LENGTHOF id
  | recordConst
  | string
  | constant
  | type
  | id
  | $
  | segmentRegister
  | register
  | ST
  | ST ( expr )

echoDir
  ECHO
  arbitraryText ;;
  %OUT arbitraryText ;;

elseifBlock
  elseifStatement ;;
  directiveList
  □ elseifBlock □

elseifStatement
  ELSEIF constExpr
  | ELSEIFE constExpr
  | ELSEIFB textItem
  | ELSEIFNB textItem
  | ELSEIFDEF id
  | ELSEIFNDEF id
  | ELSEIFDIF textItem , textItem
  | ELSEIFDIFI textItem , textItem
  | ELSEIFIDN textItem , textItem
  | ELSEIFIDNI textItem , textItem
  | ELSEIF1
  | ELSEIF2

endDir
  END □ immExpr □ ;;

endpDir
  procId ENDP ;;

endsDir
  id ENDS ;;

equDir
  textMacroId EQU equType ;;

equType
  immExpr | textLiteral

errorDir
  errorOpt ;;

errorOpt
  .ERR □ textItem □
  | .ERRE constExpr □ optText □
  | .ERRNZ constExpr □ optText □
  | .ERRB textItem □ optText □
  | .ERRNB textItem □ optText □
  | .ERRDEF id □ optText □
  | .ERRNDEF id □ optText □
  | .ERRDIF textItem , textItem □
optText □
  | .ERRDIFI textItem , textItem □
optText □
  | .ERRIDN textItem , textItem □
optText □
```

```
 | .ERRIDNI textItem , textItem □
optText □
 | .ERR1 □ textItem □
 | .ERR2 □ textItem □

exitDir
  .EXIT   □ expr □ ;;

exitmDir
 : EXITM | EXITM textItem

exponent
  E □ sign □ decNumber

expr
  SHORT e05
 | .TYPE e01
 | OPATTR e01
 | e01

exprList
  expr | exprList , expr

externDef
 □ langType □ id □ ( altId ) □ :
externType

externDir
  externKey externList ;;

externKey
  EXTRN | EXTERN | EXTERNDEF

externList
  externDef | externList , □ ;; □
externDef

externType
  ABS | qualifiedType

fieldAlign
  constExpr

fieldInit
 □ initValue □ | structInstance

fieldInitList
  fieldInit | fieldInitList , □ ;; □
fieldInit

fileChar
  delimiter

fileCharList
  fileChar | fileCharList fileChar

fileSpec
```

fileCharList | *textLiteral*

flagName
ZERO? | CARRY? | OVERFLOW? | SIGN? |
PARITY?

floatNumber
□ *sign* □ *decNumber* . □ *decNumber* □
□ *exponent* □
| *digits* R | *digits* r

forcDir
FORC | IRPC

forDir
FOR | IRP

forParm
id □ : *forParmType* □

forParmType
REQ | = *textLiteral*

fpuRegister
ST *expr*

frameExpr
SEG *id*
| DGROUP : *id*
| *segmentRegister* : *id*
| *id*

generalDir
modelDir | *segOrderDir* | *nameDir*
| *includeLibDir* | *commentDir*
| *groupDir* | *assumeDir*
| *structDir* | *recordDir* | *typedefDir*
| *externDir* | *publicDir* | *commDir* |
protoTypeDir
| *equDir* | *=Dir* | *textDir*
| *contextDir* | *optionDir* | *processorDir*
| *radixDir*
| *titleDir* | *pageDir* | *listDir*
| *crefDir* | *echoDir*
| *ifDir* | *errorDir* | *includeDir*
| *macroDir* | *macroCall* | *macroRepeat* |
purgeDir
| *macroWhile* | *macroFor* | *macroForc*
| *aliasDir*

gpRegister
AX | EAX | CX | ECX | DX | EDX | BX |
EBX | DI | EDI | SI | ESI | BP | EBP |
SP | ESP | RSP | R8W | R8D | R9W | R9D |
R12D | R13W | R13D | R14W | R14D

groupDir
  groupId GROUP *segIdList*

groupId
  id

hexdigit
  a | b | c | d | e | f | A | B | C | D | E | F

id
  The first character of the identifier can be an upper or lower-case alphabetic character (`[A-Za-z]`) or any of these four characters: `@ _ $ ?` The remaining characters can be any of those same characters or a decimal digit (`[0-9]`). Maximum length is 247 characters.

idList
  id | *idList* , *id*

ifDir
  ifStatement ;;
  directiveList
  ⟦ *elseifBlock* ⟧
  ⟦ ELSE ;;
  directiveList ⟧ ;;

ifStatement
  IF *constExpr*
  | IFE *constExpr*
  | IFB *textItem*
  | IFNB *textItem*
  | IFDEF *id*
  | IFNDEF *id*
  | IFDIF *textItem* , *textItem*
  | IFDIFI *textItem* , *textItem*
  | IFIDN *textItem* , *textItem*
  | IFIDNI *textItem* , *textItem*
  | IF1
  | IF2

immExpr
  expr

includeDir
  INCLUDE *fileSpec* ;;

includeLibDir
  INCLUDELIB *fileSpec* ;;

initValue
  immExpr
  | *string*

| `?`
| `constExpr` `DUP` (`scalarInstList`)
| `floatNumber`
| `bcdConst`

`inSegDir`
□ `labelDef` □ `inSegmentDir`

`inSegDirList`
`inSegDir` | `inSegDirList` `inSegDir`

`inSegmentDir`
`instruction`
| `dataDir`
| `controlDir`
| `startupDir`
| `exitDir`
| `offsetDir`
| `labelDir`
| `procDir` □ `localDirList` □ □
`inSegDirList` □ `endpDir`
| `invokeDir`
| `generalDir`

`instrPrefix`
`REP` | `REPE` | `REPZ` | `REPNE` | `REPNZ` |
`LOCK`

`instruction`
□ `instrPrefix` □ `asmInstruction`

`invokeArg`
`register` `::` `register` | `expr` | `ADDR`
`expr`

`invokeDir`
`INVOKE` `expr` □ `,` □ `;;` □ `invokeList`
□ `;;`

`invokeList`
`invokeArg` | `invokeList` `,` □ `;;` □
`invokeArg`

`keyword`
Any reserved word.

`keywordList`
`keyword` | `keyword` `keywordList`

`labelDef`
`id` `:` | `id` `::` | `@@:`

`labelDir`
`id` `LABEL` `qualifiedType` `;;`

`langType`

C | PASCAL | FORTRAN | BASIC | SYSCALL | STDCALL

listDir
listOption ;;

listOption
.LIST
| .NOLIST
| .XLIST
| .LISTALL
| .LISTIF
| .LFCOND
| .NOLISTIF
| .SFCOND
| .TFCOND
| .LISTMACROALL | .LALL
| .NOLISTMACRO | .SALL
| .LISTMACRO | .XALL

localDef
LOCAL *idList* ;;

localDir
LOCAL *parmList* ;;

localDirList
localDir | *localDirList* *localDir*

localList
localDef | *localList* *localDef*

macroArg
% *constExpr*
| % *textMacroId*
| % *macroFuncId* (*macroArgList*)
| *string*
| *arbitraryText*
| < *arbitraryText* >

macroArgList
macroArg | *macroArgList* , *macroArg*

macroBody
□ *localList* □ *macroStmtList*

macroCall
id *macroArgList* ;;
| *id* (*macroArgList*)

macroDir
id MACRO □ *macroParmList* □ ;;
macroBody
ENDM ;;

macroFor
forDir *forParm* , < *macroArgList* >
;;
macroBody
ENDM ;;

macroForc
forcDir *id* , *textLiteral* ;;
macroBody
ENDM ;;

macroFuncId
id

macroId
macroProcId | *macroFuncId*

macroIdList
macroId | *macroIdList* , *macroId*

macroLabel
id

macroParm
id □ : *parmType* □

macroParmList
macroParm | *macroParmList* , □ ;; □
macroParm

macroProcId
id

macroRepeat
repeatDir *constExpr* ;;
macroBody
ENDM ;;

macroStmt
directive
| *exitmDir*
| : *macroLabel*
| GOTO *macroLabel*

macroStmtList
macroStmt ;;
| *macroStmtList* *macroStmt* ;; \

macroWhile
WHILE *constExpr* ;;
macroBody
ENDM ;;

mapType
ALL | NONE | NOTPUBLIC

memOption
`TINY` | `SMALL` | `MEDIUM` | `COMPACT` | `LARGE` | `HUGE` | `FLAT`

mnemonic
Instruction name.

modelDir
`.MODEL`
memOption □ `,` *modelOptlist* □ *;;*

modelOpt
langType | *stackOption*

modelOptlist
modelOpt | *modelOptlist* `,` *modelOpt*

module
□ *directiveList* □ *endDir*

mulOp
`*` | `/` | `MOD`

nameDir
`NAME`
id *;;*

nearfar
`NEAR` | `FAR`

nestedStruct
structHdr □ *id* □ *;;*
structBody
`ENDS` *;;*

offsetDir
offsetDirType *;;*

offsetDirType
`EVEN` | `ORG` *immExpr* | `ALIGN` □ *constExpr* □

offsetType
`GROUP` | `SEGMENT` | `FLAT`

oldRecordFieldList
□ *constExpr* □ | *oldRecordFieldList* `,` □ *constExpr* □

optionDir
`OPTION` *optionList* *;;*

optionItem
`CASEMAP` : *mapType*
| `DOTNAME` | `NODOTNAME`
| `EMULATOR` | `NOEMULATOR`
| `EPILOGUE` `:` *macroId*

| EXPR16 | EXPR32
| LANGUAGE : *langType*
| LJMP | NOLJMP
| M510 | NOM510
| NOKEYWORD : < *keywordList* >
| NOSIGNEXTEND
| OFFSET : *offsetType*
| OLDMACROS | NOOLDMACROS
| OLDSTRUCTS | NOOLDSTRUCTS
| PROC : *oVisibility*
| PROLOGUE : *macroId*
| READONLY | NOREADONLY
| SCOPED | NOSCOPED
| SEGMENT : *segSize*
| SETIF2 : *bool*

optionList
optionItem | *optionList* , □ ;; □
optionItem

optText
, *textItem*

orOp
OR | XOR

oVisibility
PUBLIC | PRIVATE | EXPORT

pageDir
PAGE □ *pageExpr* □ ;;

pageExpr
+ | □ *pageLength* □ □ , *pageWidth* □

pageLength
constExpr

pageWidth
constExpr

parm
parmId □ : *qualifiedType* □ | *parmId*
□ *constExpr* □ □ : *qualifiedType* □

parmId
id

parmList
parm | *parmList* , □ ;; □ *parm*

parmType
REQ | = *textLiteral* | VARARG

pOptions
□ *distance* □ □ *langType* □ □

```
oVisibility □

primary
  expr binaryOp expr | flagName | expr

procDir
  procId PROC
  □ pOptions □ □ < macroArgList > □
  □ usesRegs □ □ procParmList □

processor
  | .386 | .386p | .486 | .486P
  | .586 | .586P | .686 | .686P | .387

processorDir
  processor ;;
  | coprocessor ;;

procId
  id

procItem
  instrPrefix | dataDir | labelDir |
offsetDir | generalDir

procParmList
  □ , □ ;; □ parmList □
  □ , □ ;; □ parmId :VARARG □

protoArg
  □ id □ : qualifiedType

protoArgList
  □ , □ ;; □ protoList □
  □ , □ ;; □ □ id □ :VARARG □

protoList
  protoArg
  | protoList , □ ;; □ protoArg

protoSpec
  □ distance □ □ langType □ □
protoArgList □| typeId

protoTypeDir
  id PROTO protoSpec

pubDef
  □ langType □ id

publicDir
  PUBLIC pubList ;;

pubList
  pubDef | pubList , □ ;; □ pubDef

purgeDir
```

PURGE *macroIdList*

qualifiedType
type | □ *distance* □ PTR □
qualifiedType □

qualifier
qualifiedType | PROTO *protoSpec*

quote
" | '

qwordRegister
RAX | RCX | RDX | RBX | RDI | RSI |
RBP | R8 | R9 | R10 | R11 | R12 | R13 |
R14 | R15

radixDir
.RADIX *constExpr* *;;*

radixOverride
h | o | q | t | y | H | O | Q | T |
Y

recordConst
recordTag { *oldRecordFieldList* } |
recordTag < *oldRecordFieldList* >

recordDir
recordTag RECORD *bitDefList* *;;*

recordFieldList
□ *constExpr* □ | *recordFieldList* , □
;; □ □ *constExpr* □

recordInstance
{ □ *;;* □ *recordFieldList* □ *;;* □ }
| < *oldRecordFieldList* >
| *constExpr* DUP (*recordInstance*)

recordInstList
recordInstance | *recordInstList* , □
;; □ *recordInstance*

recordTag
id

register
specialRegister | *gpRegister* |
byteRegister | *qwordRegister* |
fpuRegister | *SIMDRegister* |
segmentRegister

regList
register | *regList* *register*

relOp

```
  EQ | NE | LT | LE | GT | GE

repeatBlock
  .REPEAT ;;
  blockStatements ;; untilDir ;;

repeatDir
  REPEAT | REPT

scalarInstList
  initValue | scalarInstList , □ ;; □
initValue

segAlign
  BYTE | WORD | DWORD | PARA | PAGE

segAttrib
  PUBLIC | STACK | COMMON | MEMORY | AT
constExpr | PRIVATE

segDir
  .CODE
  □ segId □
  | .DATA
  | .DATA?
  | .CONST
  | .FARDATA □ segId □
  | .FARDATA? □ segId □
  | .STACK □ constExpr □

segId
  id

segIdList
  segId
  | segIdList , segId

segmentDef
  segmentDir □ inSegDirList □ endsDir |
simpleSegDir □ inSegDirList □ □ endsDir
□

segmentDir
  segId SEGMENT □ segOptionList □ ;;

segmentRegister
  CS | DS | ES | FS | GS | SS

segOption
  segAlign
  | segRO
  | segAttrib
  | segSize
  | className

segOptionList
```

segOption | *segOptionList* *segOption*

segOrderDir
.ALPHA | .SEQ | .DOSSEG | DOSSEG

segRO
READONLY

segSize
USE16 | USE32 | FLAT

shiftOp
SHR | SHL

sign
+ | -

simdRegister
MM0 | MM1 | MM2 | MM3 | MM4 | MM5 | MM6 | MM7
| *xmmRegister*
| YMM0 | YMM1 | YMM2 | YMM3 | YMM4 | YMM5 | YMM6 | YMM7 | YMM8 | YMM9 | YMM10 | YMM11 | YMM12 | YMM13 | YMM14 | YMM15

simpleExpr
(*cExpr*) | *primary*

simpleSegDir
segDir *;;*

sizeArg
id | *type* | *e10*

specialChars
: | . | [|] | (|) | < | > | { | }
| + | - | / | * | & | % | !
| ' | \ | = | ; | , | "
| *whiteSpaceCharacter*
| *endOfLine*

specialRegister
CR0 | CR2 | CR3 | DR0 | DR1 | DR2 | DR3 | DR6 | DR7 | TR3 | TR4 | TR5 | TR6 | TR7

stackOption
NEARSTACK | FARSTACK

startupDir
.STARTUP *;;*

stext
stringChar | *stext* *stringChar*

`string`
  `quote` □ `stext` □ `quote`

`stringChar`
  `quote` `quote` | Any character except quote.

`structBody`
  `structItem` `;;`
  | `structBody` `structItem` `;;`

`structDir`
  `structTag` `structHdr` □ `fieldAlign` □
  □ `,` `NONUNIQUE` □ `;;`
  `structBody`
  `structTag`
  `ENDS` `;;`

`structHdr`
  `STRUC` | `STRUCT` | `UNION`

`structInstance`
  `<` □ `fieldInitList` □ `>`
  | `{` □ `;;` □ □ `fieldInitList` □ □ `;;` □ `}`
  | `constExpr` `DUP` (`structInstList`)

`structInstList`
  `structInstance` | `structInstList` `,` □ `;;` □ `structInstance`

`structItem`
  `dataDir`
  | `generalDir`
  | `offsetDir`
  | `nestedStruct`

`structTag`
  `id`

`term`
  `simpleExpr` | `!` `simpleExpr`

`text`
  `textLiteral` | `text` character | `!` `character` `text` | `character` | `!` `character`

`textDir`
  `id` `textMacroDir` `;;`

`textItem`
  `textLiteral` | `textMacroId` | `%` `constExpr`

`textLen`

constExpr

textList

textItem | *textList* , □ ;; □

textItem

textLiteral

< *text* > ;;

textMacroDir

CATSTR □ *textList* □

| TEXTEQU □ *textList* □

| SIZESTR *textItem*

| SUBSTR *textItem* , *textStart* □ ,

textLen □

| INSTR □ *textStart* , □ *textItem* ,

textItem

textMacroId

id

textStart

constExpr

titleDir

titleType *arbitraryText* ;;

titleType

TITLE | SUBTITLE | SUBTTL

type

structTag

| *unionTag*

| *recordTag*

| *distance*

| *dataType*

| *typeId*

typedefDir

typeId TYPEDEF qualifier

typeId

id

unionTag

id

untilDir

.UNTIL *cExpr* ;;

.UNTILCXZ □ *cxzExpr* □ ;;

usesRegs

USES *regList*

whileBlock

.WHILE

`cExpr` `;;`
`blockStatements` `;;`
`.ENDW`

`whiteSpaceCharacter`
ASCII 8, 9, 11–13, 26, 32

`xmmRegister`
`XMM0` | `XMM1` | `XMM2` | `XMM3` | `XMM4` |
`XMM5` | `XMM6` | `XMM7` | `XMM8` | `XMM9` |
`XMM10` | `XMM11` | `XMM12` | `XMM13` | `XMM14` |
`XMM15`