Linux Ubuntu 22.4.5 Manual Pages on command 'nasm.1'

$ man nasm.1

NASM(1) The Netwide Assembler Project NASM(1)

NAME

nasm - the Netwide Assembler, a portable 80x86 assembler

SYNOPSIS

nasm [-@ response file] [-f format] [-o outfile] [-l listfile] [options...]

filename

DESCRIPTION

The nasm command assembles the file filename and directs output to the file outfile if specified. If outfile is not specified, nasm will derive a default output file name from the name of its input file, usually by appending ?.o? or ?.obj?, or by removing all extensions for a raw binary file. Failing that, the output file name will be ?nasm.out?.

OPTIONS

-@ filename

Causes nasm to process options from filename as if they were included on the command line.

-a

Causes nasm to assemble the given input file without first applying the macro preprocessor.

-D|-d macro[=value]

Pre-defines a single-line macro.

-E|-e

Causes nasm to preprocess the given input file, and write the output to stdout (or the specified output file name), and not actually assemble anything.

-f format

Specifies the output file format. To see a list of valid output formats, use the -hf option.

-F format

Specifies the debug information format. To see a list of valid output formats, use the -y option (for example -felf -y).

-g

Causes nasm to generate debug information.

-gformat

Equivalent to -g -F format.

-h

Causes nasm to exit immediately, after giving a summary of its invocation options.

-hf

Same as -h , but also lists all valid output formats.

-I|-i directory

Adds a directory to the search path for include files. The directory specification must include the trailing slash, as it will be directly prepended to the name of the include file.

-l listfile

Causes an assembly listing to be directed to the given file, in which the original source is displayed on the right hand side (plus the source for included files and the expansions of multi-line macros) and the generated code is shown in hex on the left.

-M

Causes nasm to output Makefile-style dependencies to stdout; normal output is suppressed.

-MG file

Same as -M but assumes that missing Makefile dependecies are generated and added to dependency list without a prefix.

-MF file

Output Makefile-style dependencies to the specified file.

-MD file

Same as a combination of -M and -MF options.

-MT file

Override the default name of the dependency target dependency target name. This is normally the same as the output filename, specified by the -o option.

-MQ file

The same as -MT except it tries to quote characters that have special meaning in Makefile syntax. This is not foolproof, as not all characters with special meaning are quotable in Make.

-MP

Emit phony target.

-O number

Optimize branch offsets.

? -O0: No optimization

? -O1: Minimal optimization

? -Ox: Multipass optimization (default)

-o outfile

Specifies a precise name for the output file, overriding nasm's default means of determining it.

-P|-p file

Specifies a file to be pre-included, before the main source file starts to be processed.

-s

Causes nasm to send its error messages and/or help text to stdout instead of stderr.

-t

Causes nasm to assemble in SciTech TASM compatible mode.

-U|-u macro

Undefines a single-line macro.

-v

Causes nasm to exit immediately, after displaying its version number.

*-W[no-]foo'

Causes nasm to enable or disable certain classes of warning messages, in gcc-like style, for example -Worphan-labels or -Wno-orphan-labels.

-w[+-]foo

Causes nasm to enable or disable certain classes of warning messages, for example -w+orphan-labels or -w-macro-params.

-X format

Specifies error reporting format (gnu or vc).

-y

Causes nasm to list supported debug formats.

-Z filename

Causes nasm to redirect error messages to filename. This option exists to support operating systems on which stderr is not easily redirected.

--prefix, --postfix

Prepend or append (respectively) the given argument to all global or extern variables.

SYNTAX

This man page does not fully describe the syntax of nasm's assembly language, but does give a summary of the differences from other assemblers.

Registers have no leading ?%? sign, unlike gas, and floating-point stack registers are referred to as st0, st1, and so on.

Floating-point instructions may use either the single-operand form or the double. A TO keyword is provided; thus, one could either write

fadd st0,st1

fadd st1,st0

or one could use the alternative single-operand forms

fadd st1

fadd to st1

Uninitialised storage is reserved using the RESB, RESW, RESD, RESQ, REST and RESO pseudo-opcodes, each taking one parameter which gives the number of bytes, words, doublewords, quadwords or ten-byte words to reserve.

Repetition of data items is not done by the DUP keyword as seen in DOS assemblers, but by the use of the TIMES prefix, like this:

message: times 3 db 'abc'

```
        times 64-$+message db 0
```

which defines the string abcabcabc, followed by the right number of zero bytes to make the total length up to 64 bytes.

Symbol references are always understood to be immediate (i.e. the address of the symbol), unless square brackets are used, in which case the contents of the memory location are used. Thus:

```
    mov ax,wordvar
```

loads AX with the address of the variable wordvar, whereas

```
    mov ax,[wordvar]
    mov ax,[wordvar+1]
    mov ax,[es:wordvar+bx]
```

all refer to the contents of memory locations. The syntaxes

```
    mov ax,es:wordvar[bx]
    es mov ax,wordvar[1]
```

are not legal at all, although the use of a segment register name as an instruction prefix is valid, and can be used with instructions such as LODSB which can?t be overridden any other way.

Constants may be expressed numerically in most formats: a trailing H, Q or B denotes hex, octal or binary respectively, and a leading ?0x? or ?$? denotes hex as well. Leading zeros are not treated specially at all. Character constants may be enclosed in single or double quotes; there is no escape character. The ordering is little-endian (reversed), so that the character constant 'abcd' denotes 0x64636261 and not 0x61626364.

Local labels begin with a period, and their ?locality? is granted by the assembler prepending the name of the previous non-local symbol. Thus declaring a label ?.loop? after a label ?label? has actually defined a symbol called ?label.loop?.

DIRECTIVES

SECTION name or SEGMENT name causes nasm to direct all following code to the named section. Section names vary with output file format, although most formats support the names .text, .data and .bss. (The exception is the obj format, in which all segments are user-definable.)

ABSOLUTE address causes nasm to position its notional assembly point at an absolute address: so no code or data may be generated, but you can use RESB, RESW and RESD

to move the assembly point further on, and you can define labels. So this directive may be used to define data structures. When you have finished doing absolute assembly, you must issue another SECTION directive to return to normal assembly.

BITS 16, BITS 32 or BITS 64 switches the default processor mode for which nasm is generating code: it is equivalent to USE16 or USE32 in DOS assemblers.

EXTERN symbol and GLOBAL symbol import and export symbol definitions, respectively, from and to other modules. Note that the GLOBAL directive must appear before the definition of the symbol it refers to.

STRUC strucname and ENDSTRUC, when used to bracket a number of RESB, RESW or similar instructions, define a data structure. In addition to defining the offsets of the structure members, the construct also defines a symbol for the size of the structure, which is simply the structure name with size tacked on to the end.

FORMAT-SPECIFIC DIRECTIVES

ORG address is used by the bin flat-form binary output format, and specifies the address at which the output code will eventually be loaded.

GROUP grpname seg1 seg2... is used by the obj (Microsoft 16-bit) output format, and defines segment groups. This format also uses UPPERCASE, which directs that all segment, group and symbol names output to the object file should be in uppercase. Note that the actual assembly is still case sensitive.

LIBRARY libname is used by the rdf output format, and causes a dependency record to be written to the output file which indicates that the program requires a certain library in order to run.

MACRO PREPROCESSOR

Single-line macros are defined using the %define or %idefine commands, in a similar fashion to the C preprocessor. They can be overloaded with respect to number of parameters, although defining a macro with no parameters prevents the definition of any macro with the same name taking parameters, and vice versa. %define defines macros whose names match case-sensitively, whereas %idefine defines case-insensitive macros.

Multi-line macros are defined using %macro and %imacro (the distinction is the same as that between %define and %idefine), whose syntax is as follows

```
%macro name minprm[-maxprm][+][.nolist] [defaults]
    <some lines of macro expansion text>
```

%endmacro

Again, these macros may be overloaded. The trailing plus sign indicates that any parameters after the last one get subsumed, with their separating commas, into the last parameter. The defaults part can be used to specify defaults for unspecified macro parameters after minparam. %endm is a valid synonym for %endmacro.

To refer to the macro parameters within a macro expansion, you use %1, %2 and so on. You can also enforce that a macro parameter should contain a condition code by using %+1, and you can invert the condition code by using %-1. You can also define a label specific to a macro invocation by prefixing it with a double ?%? sign.

Files can be included using the %include directive, which works like C.

The preprocessor has a ?context stack?, which may be used by one macro to store information that a later one will retrieve. You can push a context on the stack using %push, remove one using %pop, and change the name of the top context (without disturbing any associated definitions) using %repl. Labels and %define macros specific to the top context may be defined by prefixing their names with %$, and things specific to the next context down with %$$, and so on.

Conditional assembly is done by means of %ifdef, %ifndef, %else and %endif as in C. (Except that %ifdef can accept several putative macro names, and will evaluate TRUE if any of them is defined.) In addition, the directives %ifctx and %ifnctx can be used to condition on the name of the top context on the context stack. The obvious set of ?else-if? directives, %elifdef, %elifndef, %elifctx and %elifnctx are also supported.

BUGS

Please report bugs through the bug tracker function at http://nasm.us.

SEE ALSO

as(1), ld(1).