# NASM — The Netwide Assembler

***version 0.99.05−20071023***

© 2007 The NASM Development Team

All rights reserved. This document is redistributable under the license given in the file "COPYING" distributed in the NASM archive.

# Contents

Chapter 1: Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
  1.1 What Is NASM? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
    1.1.1 Why Yet Another Assembler? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
    1.1.2 License Conditions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
  1.2 Contact Information . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
  1.3 Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11
    1.3.1 Installing NASM under MS−DOS or Windows . . . . . . . . . . . . . . . . . . . . . .11
    1.3.2 Installing NASM under Unix . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11
Chapter 2: Running NASM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .13
  2.1 NASM Command−Line Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .13
    2.1.1 The `−o` Option: Specifying the Output File Name . . . . . . . . . . . . . . . . . . . .13
    2.1.2 The `−f` Option: Specifying the Output File Format . . . . . . . . . . . . . . . . . . .14
    2.1.3 The `−l` Option: Generating a Listing File . . . . . . . . . . . . . . . . . . . . . . .14
    2.1.4 The `−M` Option: Generate Makefile Dependencies . . . . . . . . . . . . . . . . . . . .14
    2.1.5 The `−MG` Option: Generate Makefile Dependencies . . . . . . . . . . . . . . . . . . .14
    2.1.6 The `−F` Option: Selecting a Debug Information Format . . . . . . . . . . . . . . . . .14
    2.1.7 The `−g` Option: Enabling Debug Information . . . . . . . . . . . . . . . . . . . . . .15
    2.1.8 The `−X` Option: Selecting an Error Reporting Format . . . . . . . . . . . . . . . . . .15
    2.1.9 The `−Z` Option: Send Errors to a File . . . . . . . . . . . . . . . . . . . . . . . . .15
    2.1.10 The `−s` Option: Send Errors to `stdout` . . . . . . . . . . . . . . . . . . . . . . .15
    2.1.11 The `−i` Option: Include File Search Directories . . . . . . . . . . . . . . . . . . .15
    2.1.12 The `−p` Option: Pre−Include a File . . . . . . . . . . . . . . . . . . . . . . . . . .16
    2.1.13 The `−d` Option: Pre−Define a Macro . . . . . . . . . . . . . . . . . . . . . . . . .16
    2.1.14 The `−u` Option: Undefine a Macro . . . . . . . . . . . . . . . . . . . . . . . . . . .16
    2.1.15 The `−E` Option: Preprocess Only . . . . . . . . . . . . . . . . . . . . . . . . . . .17
    2.1.16 The `−a` Option: Don’t Preprocess At All . . . . . . . . . . . . . . . . . . . . . . .17
    2.1.17 The `−On` Option: Specifying Multipass Optimization . . . . . . . . . . . . . . . . . .17
    2.1.18 The `−t` option: Enable TASM Compatibility Mode . . . . . . . . . . . . . . . . . . . .17
    2.1.19 The `−w` Option: Enable or Disable Assembly Warnings . . . . . . . . . . . . . . . . .18
    2.1.20 The `−v` Option: Display Version Info . . . . . . . . . . . . . . . . . . . . . . . . .18

2.1.21 The `−y` Option: Display Available Debug Info Formats 19
2.1.22 The `−−prefix` and `−−postfix` Options. 19
2.1.23 The `NASMENV` Environment Variable 19
2.2 Quick Start for MASM Users 19
2.2.1 NASM Is Case−Sensitive 19
2.2.2 NASM Requires Square Brackets For Memory References 19
2.2.3 NASM Doesn’t Store Variable Types 20
2.2.4 NASM Doesn’t `ASSUME` 20
2.2.5 NASM Doesn’t Support Memory Models 20
2.2.6 Floating−Point Differences 20
2.2.7 Other Differences 21
Chapter 3: The NASM Language 22
3.1 Layout of a NASM Source Line 22
3.2 Pseudo−Instructions 23
3.2.1 `DB` and friends: Declaring initialized Data 23
3.2.2 `RESB` and friends: Declaring Uninitialized Data 23
3.2.3 `INCBIN`: Including External Binary Files 23
3.2.4 `EQU`: Defining Constants. 24
3.2.5 `TIMES`: Repeating Instructions or Data 24
3.3 Effective Addresses 24
3.4 Constants. 25
3.4.1 Numeric Constants 25
3.4.2 Character Constants 26
3.4.3 String Constants 26
3.4.4 Floating−Point Constants 26
3.5 Expressions 27
3.5.1 `|`: Bitwise OR Operator 27
3.5.2 `^`: Bitwise XOR Operator 27
3.5.3 `&`: Bitwise AND Operator 27
3.5.4 `<<` and `>>`: Bit Shift Operators 28
3.5.5 `+` and `−`: Addition and Subtraction Operators 28
3.5.6 `*`, `/`, `//`, `%` and `%%`: Multiplication and Division 28
3.5.7 Unary Operators: `+`, `−`, `~`, `!` and `SEG` 28
3.6 `SEG` and `WRT` 28
3.7 `STRICT`: Inhibiting Optimization 29

3.8 Critical Expressions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .29
3.9 Local Labels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .30
Chapter 4: The NASM Preprocessor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .32
4.1 Single−Line Macros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .32
4.1.1 The Normal Way: `%define` . . . . . . . . . . . . . . . . . . . . . . . . . . .32
4.1.2 Enhancing %define: `%xdefine` . . . . . . . . . . . . . . . . . . . . . . . . .33
4.1.3 Concatenating Single Line Macro Tokens: `%+` . . . . . . . . . . . . . . . . . . .34
4.1.4 Undefining macros: `%undef` . . . . . . . . . . . . . . . . . . . . . . . . . . .34
4.1.5 Preprocessor Variables: `%assign` . . . . . . . . . . . . . . . . . . . . . . . .35
4.2 String Handling in Macros: `%strlen` and `%substr` . . . . . . . . . . . . . . . . . .35
4.2.1 String Length: `%strlen` . . . . . . . . . . . . . . . . . . . . . . . . . . . . .35
4.2.2 Sub−strings: `%substr` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .35
4.3 Multi−Line Macros: `%macro` . . . . . . . . . . . . . . . . . . . . . . . . . . . . .36
4.3.1 Overloading Multi−Line Macros . . . . . . . . . . . . . . . . . . . . . . . . . .36
4.3.2 Macro−Local Labels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .37
4.3.3 Greedy Macro Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . .37
4.3.4 Default Macro Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . .38
4.3.5 `%0`: Macro Parameter Counter . . . . . . . . . . . . . . . . . . . . . . . . . .39
4.3.6 `%rotate`: Rotating Macro Parameters . . . . . . . . . . . . . . . . . . . . . . .39
4.3.7 Concatenating Macro Parameters . . . . . . . . . . . . . . . . . . . . . . . . . .40
4.3.8 Condition Codes as Macro Parameters . . . . . . . . . . . . . . . . . . . . . . .41
4.3.9 Disabling Listing Expansion . . . . . . . . . . . . . . . . . . . . . . . . . . . .41
4.4 Conditional Assembly . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .41
4.4.1 `%ifdef`: Testing Single−Line Macro Existence . . . . . . . . . . . . . . . . . . .42
4.4.2 `ifmacro`: Testing Multi−Line Macro Existence . . . . . . . . . . . . . . . . . . .42
4.4.3 `%ifctx`: Testing the Context Stack . . . . . . . . . . . . . . . . . . . . . . . .42
4.4.4 `%if`: Testing Arbitrary Numeric Expressions . . . . . . . . . . . . . . . . . . .43
4.4.5 `%ifidn` and `%ifidni`: Testing Exact Text Identity . . . . . . . . . . . . . . . .43
4.4.6 `%ifid`, `%ifnum`, `%ifstr`: Testing Token Types . . . . . . . . . . . . . . . . . .43
4.4.7 `%error`: Reporting User−Defined Errors . . . . . . . . . . . . . . . . . . . . . .44
4.5 Preprocessor Loops: `%rep` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .45
4.6 Including Other Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .45
4.7 The Context Stack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .46
4.7.1 `%push` and `%pop`: Creating and Removing Contexts . . . . . . . . . . . . . . . . .46
4.7.2 Context−Local Labels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .46

4.7.3 Context−Local Single−Line Macros. . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.7.4 `%repl`: Renaming a Context . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.7.5 Example Use of the Context Stack: Block IFs . . . . . . . . . . . . . . . . . . . . . . 47

4.8 Standard Macros. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

4.8.1 `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `___NASM_PATCHLEVEL__`: NASM Version. . . . . . . . . . . . . . . . . . . . . . . . 49

4.8.2 `__NASM_VERSION_ID__`: NASM Version ID. . . . . . . . . . . . . . . . . . . . . 49

4.8.3 `__NASM_VER__`: NASM Version string. . . . . . . . . . . . . . . . . . . . . . . . 49

4.8.4 `__FILE__` and `__LINE__`: File Name and Line Number . . . . . . . . . . . . . . . . 49

4.8.5 `__BITS__`: Current BITS Mode . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.8.6 `STRUC` and `ENDSTRUC`: Declaring Structure Data Types . . . . . . . . . . . . . . . . . 50

4.8.7 `ISTRUC`, `AT` and `IEND`: Declaring Instances of Structures . . . . . . . . . . . . . . . 51

4.8.8 `ALIGN` and `ALIGNB`: Data Alignment . . . . . . . . . . . . . . . . . . . . . . . . 51

4.9 TASM Compatible Preprocessor Directives . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.9.1 `%arg` Directive. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.9.2 `%stacksize` Directive. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

4.9.3 `%local` Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

4.10 Other Preprocessor Directives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.10.1 `%line` Directive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.10.2 `%!<env>`: Read an environment variable. . . . . . . . . . . . . . . . . . . . . . . 54

Chapter 5: Assembler Directives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

5.1 `BITS`: Specifying Target Processor Mode . . . . . . . . . . . . . . . . . . . . . . . . 56

5.1.1 `USE16` & `USE32`: Aliases for BITS . . . . . . . . . . . . . . . . . . . . . . . . . 57

5.2 `DEFAULT`: Change the assembler defaults. . . . . . . . . . . . . . . . . . . . . . . . 57

5.3 `SECTION` or `SEGMENT`: Changing and Defining Sections . . . . . . . . . . . . . . . . . 57

5.3.1 The `__SECT__` Macro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

5.4 `ABSOLUTE`: Defining Absolute Labels . . . . . . . . . . . . . . . . . . . . . . . . . 58

5.5 `EXTERN`: Importing Symbols from Other Modules . . . . . . . . . . . . . . . . . . . . . 59

5.6 `GLOBAL`: Exporting Symbols to Other Modules. . . . . . . . . . . . . . . . . . . . . . 59

5.7 `COMMON`: Defining Common Data Areas . . . . . . . . . . . . . . . . . . . . . . . . 60

5.8 `CPU`: Defining CPU Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

5.9 `FLOAT`: Handling of floating−point constants . . . . . . . . . . . . . . . . . . . . . . 61

Chapter 6: Output Formats. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

6.1 `bin`: Flat−Form Binary Output. . . . . . . . . . . . . . . . . . . . . . . . . . . . 62

6.1.1 `ORG`: Binary File Program Origin . . . . . . . . . . . . . . . . . . . . . . . . . . 62

6.1.2 `bin` Extensions to the `SECTION` Directive. . . . . . . . . . . . . . . . . . . . . . . . . .62
6.1.3 `Multisection` support for the BIN format. . . . . . . . . . . . . . . . . . . . . . . .63
6.1.4 Map files . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .63
6.2 `obj`: Microsoft OMF Object Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . .63
6.2.1 `obj` Extensions to the `SEGMENT` Directive. . . . . . . . . . . . . . . . . . . . . . . . .64
6.2.2 `GROUP`: Defining Groups of Segments . . . . . . . . . . . . . . . . . . . . . . . . . .65
6.2.3 `UPPERCASE`: Disabling Case Sensitivity in Output . . . . . . . . . . . . . . . . . . . .65
6.2.4 `IMPORT`: Importing DLL Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . .66
6.2.5 `EXPORT`: Exporting DLL Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . .66
6.2.6 `..start`: Defining the Program Entry Point . . . . . . . . . . . . . . . . . . . . . . .66
6.2.7 `obj` Extensions to the `EXTERN` Directive . . . . . . . . . . . . . . . . . . . . . . . .67
6.2.8 `obj` Extensions to the `COMMON` Directive . . . . . . . . . . . . . . . . . . . . . . . .67
6.3 `win32`: Microsoft Win32 Object Files. . . . . . . . . . . . . . . . . . . . . . . . . . . .68
6.3.1 `win32` Extensions to the `SECTION` Directive . . . . . . . . . . . . . . . . . . . . . .68
6.4 `win64`: Microsoft Win64 Object Files. . . . . . . . . . . . . . . . . . . . . . . . . . . .69
6.5 `coff`: Common Object File Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . .69
6.6 `macho`: Mach Object File Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . .69
6.7 `elf, elf32, and elf64`: Executable and Linkable Format Object Files . . . . . . . .69
6.7.1 `elf` Extensions to the `SECTION` Directive. . . . . . . . . . . . . . . . . . . . . . . . .69
6.7.2 Position−Independent Code: `elf` Special Symbols and `WRT` . . . . . . . . . . . . . . .70
6.7.3 `elf` Extensions to the `GLOBAL` Directive . . . . . . . . . . . . . . . . . . . . . . . .70
6.7.4 `elf` Extensions to the `COMMON` Directive . . . . . . . . . . . . . . . . . . . . . . . .71
6.7.5 16−bit code and ELF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .71
6.8 `aout`: Linux `a.out` Object Files . . . . . . . . . . . . . . . . . . . . . . . . . . . . .71
6.9 `aoutb`: NetBSD/FreeBSD/OpenBSD `a.out` Object Files . . . . . . . . . . . . . . . . . .71
6.10 `as86`: Minix/Linux `as86` Object Files. . . . . . . . . . . . . . . . . . . . . . . . . . .72
6.11 `rdf`: Relocatable Dynamic Object File Format . . . . . . . . . . . . . . . . . . . . . .72
6.11.1 Requiring a Library: The `LIBRARY` Directive . . . . . . . . . . . . . . . . . . . . . .72
6.11.2 Specifying a Module Name: The `MODULE` Directive . . . . . . . . . . . . . . . . . . .72
6.11.3 `rdf` Extensions to the `GLOBAL` directive . . . . . . . . . . . . . . . . . . . . . . . .73
6.11.4 `rdf` Extensions to the `EXTERN` directive . . . . . . . . . . . . . . . . . . . . . . . .73
6.12 `dbg`: Debugging Format. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .73
Chapter 7: Writing 16−bit Code (DOS, Windows 3/3.1) . . . . . . . . . . . . . . . . . . . . . .75
7.1 Producing `.EXE` Files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .75
7.1.1 Using the `obj` Format To Generate `.EXE` Files . . . . . . . . . . . . . . . . . . . . .75

7.1.2 Using the `bin` Format To Generate `.EXE` Files . . . . . . . . . . . . . . . . . . . . . . .76
7.2 Producing `.COM` Files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .77
7.2.1 Using the `bin` Format To Generate `.COM` Files . . . . . . . . . . . . . . . . . . . . . . .77
7.2.2 Using the `obj` Format To Generate `.COM` Files . . . . . . . . . . . . . . . . . . . . . . .78
7.3 Producing `.SYS` Files. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .78
7.4 Interfacing to 16−bit C Programs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .78
7.4.1 External Symbol Names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .78
7.4.2 Memory Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .79
7.4.3 Function Definitions and Function Calls . . . . . . . . . . . . . . . . . . . . . . . . . . .80
7.4.4 Accessing Data Items . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .82
7.4.5 `c16.mac`: Helper Macros for the 16−bit C Interface . . . . . . . . . . . . . . . . . . .83
7.5 Interfacing to Borland Pascal Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . .84
7.5.1 The Pascal Calling Convention . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .84
7.5.2 Borland Pascal Segment Name Restrictions . . . . . . . . . . . . . . . . . . . . . . . . .85
7.5.3 Using `c16.mac` With Pascal Programs . . . . . . . . . . . . . . . . . . . . . . . . . . .86
Chapter 8: Writing 32−bit Code (Unix, Win32, DJGPP). . . . . . . . . . . . . . . . . . . . . . . .87
8.1 Interfacing to 32−bit C Programs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .87
8.1.1 External Symbol Names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .87
8.1.2 Function Definitions and Function Calls . . . . . . . . . . . . . . . . . . . . . . . . . . .87
8.1.3 Accessing Data Items . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .89
8.1.4 `c32.mac`: Helper Macros for the 32−bit C Interface . . . . . . . . . . . . . . . . . . .89
8.2 Writing NetBSD/FreeBSD/OpenBSD and Linux/ELF Shared Libraries . . . . . . . . . . . .90
8.2.1 Obtaining the Address of the GOT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .90
8.2.2 Finding Your Local Data Items . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .91
8.2.3 Finding External and Common Data Items . . . . . . . . . . . . . . . . . . . . . . . . .92
8.2.4 Exporting Symbols to the Library User . . . . . . . . . . . . . . . . . . . . . . . . . . .92
8.2.5 Calling Procedures Outside the Library . . . . . . . . . . . . . . . . . . . . . . . . . . .93
8.2.6 Generating the Library File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .93
Chapter 9: Mixing 16 and 32 Bit Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .94
9.1 Mixed−Size Jumps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .94
9.2 Addressing Between Different−Size Segments . . . . . . . . . . . . . . . . . . . . . . . . . .94
9.3 Other Mixed−Size Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .95
Chapter 10: Writing 64−bit Code (Unix, Win64). . . . . . . . . . . . . . . . . . . . . . . . . . . .97
10.1 Immediates and displacements in 64−bit mode. . . . . . . . . . . . . . . . . . . . . . . . .97
10.2 Interfacing to 64−bit C Programs (Unix) . . . . . . . . . . . . . . . . . . . . . . . . . . . .98

10.3 Interfacing to 64−bit C Programs (Win64) . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
Chapter 11: Troubleshooting. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
11.1 Common Problems. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
11.1.1 NASM Generates Inefficient Code. . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
11.1.2 My Jumps are Out of Range . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
11.1.3 `ORG` Doesn’t Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
11.1.4 `TIMES` Doesn’t Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
11.2 Bugs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
Appendix A: Ndisasm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.2 Getting Started: Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.3 Running NDISASM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.3.1 COM Files: Specifying an Origin. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.3.2 Code Following Data: Synchronisation. . . . . . . . . . . . . . . . . . . . . . . . . . 102
A.3.3 Mixed Code and Data: Automatic (Intelligent) Synchronisation . . . . . . . . . . . . 103
A.3.4 Other Options . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
A.4 Bugs and Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

# Chapter 1: Introduction

## 1.1 What Is NASM?

The Netwide Assembler, NASM, is an 80x86 and x86−64 assembler designed for portability and modularity. It supports a range of object file formats, including Linux and `*BSD` `a.out`, `ELF`, `COFF`, `Mach−O`, Microsoft 16−bit `OBJ`, `Win32` and `Win64`. It will also output plain binary files. Its syntax is designed to be simple and easy to understand, similar to Intel's but less complex. It supports from the upto and including `Pentium`, `P6`, `MMX`, `3DNow!`, `SSE`, `SSE2`, `SSE3` and `x64` opcodes. NASM has a strong support for macro conventions.

### 1.1.1 Why Yet Another Assembler?

The Netwide Assembler grew out of an idea on `comp.lang.asm.x86` (or possibly `alt.lang.asm` – I forget which), which was essentially that there didn't seem to be a good *free* x86−series assembler around, and that maybe someone ought to write one.

- `a86` is good, but not free, and in particular you don't get any 32−bit capability until you pay. It's DOS only, too.
- `gas` is free, and ports over to DOS and Unix, but it's not very good, since it's designed to be a back end to `gcc`, which always feeds it correct code. So its error checking is minimal. Also, its syntax is horrible, from the point of view of anyone trying to actually *write* anything in it. Plus you can't write 16−bit code in it (properly.)
- `as86` is specific to Minix and Linux, and (my version at least) doesn't seem to have much (or any) documentation.
- `MASM` isn't very good, and it's (was) expensive, and it runs only under DOS.
- `TASM` is better, but still strives for MASM compatibility, which means millions of directives and tons of red tape. And its syntax is essentially MASM's, with the contradictions and quirks that entails (although it sorts out some of those by means of Ideal mode.) It's expensive too. And it's DOS−only.

So here, for your coding pleasure, is NASM. At present it's still in prototype stage – we don't promise that it can outperform any of these assemblers. But please, *please* send us bug reports, fixes, helpful information, and anything else you can get your hands on (and thanks to the many people who've done this already! You all know who you are), and we'll improve it out of all recognition. Again.

### 1.1.2 License Conditions

Please see the file `COPYING`, supplied as part of any NASM distribution archive, for the license conditions under which you may use NASM. NASM is now under the so−called GNU Lesser General Public License, LGPL.

## 1.2 Contact Information

The current version of NASM (since about 0.98.08) is maintained by a team of developers, accessible through the `nasm−devel` mailing list (see below for the link). If you want to report a bug, please read section 11.2 first.

NASM has a WWW page at `http://nasm.sourceforge.net`. If it's not there, google for us!

The original authors are e−mailable as `jules@dsf.org.uk` and `anakin@pobox.com`. The latter is no longer involved in the development team.

New releases of NASM are uploaded to the official sites `http://nasm.sourceforge.net` and to `ftp.kernel.org` and `ibiblio.org`.

Announcements are posted to `comp.lang.asm.x86`, `alt.lang.asm` and `comp.os.linux.announce`

If you want information about NASM beta releases, and the current development status, please subscribe to the `nasm−devel` email list by registering at `http://sourceforge.net/projects/nasm`.

## 1.3 Installation

### 1.3.1 Installing NASM under MS−DOS or Windows

Once you've obtained the DOS archive for NASM, `nasmXXX.zip` (where `XXX` denotes the version number of NASM contained in the archive), unpack it into its own directory (for example `c:\nasm`).

The archive will contain four executable files: the NASM executable files `nasm.exe` and `nasmw.exe`, and the NDISASM executable files `ndisasm.exe` and `ndisasmw.exe`. In each case, the file whose name ends in `w` is a `Win32` executable, designed to run under `Windows 95` or `Windows NT` Intel, and the other one is a 16−bit `DOS` executable.

The only file NASM needs to run is its own executable, so copy (at least) one of `nasm.exe` and `nasmw.exe` to a directory on your PATH, or alternatively edit `autoexec.bat` to add the `nasm` directory to your `PATH`. (If you're only installing the `Win32` version, you may wish to rename it to `nasm.exe`.)

That's it – NASM is installed. You don't need the nasm directory to be present to run NASM (unless you've added it to your `PATH`), so you can delete it if you need to save space; however, you may want to keep the documentation or test programs.

If you've downloaded the DOS source archive, `nasmXXXs.zip`, the `nasm` directory will also contain the full NASM source code, and a selection of Makefiles you can (hopefully) use to rebuild your copy of NASM from scratch.

Note that the source files `insnsa.c`, `insnsd.c`, `insnsi.h` and `insnsn.c` are automatically generated from the master instruction table `insns.dat` by a Perl script; the file `macros.c` is generated from `standard.mac` by another Perl script. Although the NASM source distribution includes these generated files, you will need to rebuild them (and hence, will need a Perl interpreter) if you change insns.dat, standard.mac or the documentation. It is possible future source distributions may not include these files at all. Ports of Perl for a variety of platforms, including DOS and Windows, are available from www.cpan.org.

### 1.3.2 Installing NASM under Unix

Once you've obtained the Unix source archive for NASM, `nasm−X.XX.tar.gz` (where `X.XX` denotes the version number of NASM contained in the archive), unpack it into a directory such as `/usr/local/src`. The archive, when unpacked, will create its own subdirectory `nasm−X.XX`.

NASM is an auto−configuring package: once you've unpacked it, `cd` to the directory it's been unpacked into and type `./configure`. This shell script will find the best C compiler to use for building NASM and set up Makefiles accordingly.

Once NASM has auto−configured, you can type `make` to build the `nasm` and `ndisasm` binaries, and then `make install` to install them in `/usr/local/bin` and install the man pages `nasm.1` and `ndisasm.1` in `/usr/local/man/man1`. Alternatively, you can give options such as `--prefix` to the configure script (see the file `INSTALL` for more details), or install the programs yourself.

NASM also comes with a set of utilities for handling the `RDOFF` custom object−file format, which are in the `rdoff` subdirectory of the NASM archive. You can build these with `make rdf` and install them with `make rdf_install`, if you want them.

If NASM fails to auto−configure, you may still be able to make it compile by using the fall−back Unix makefile `Makefile.unx`. Copy or rename that file to `Makefile` and try typing `make`. There is also a Makefile.unx file in the `rdoff` subdirectory.

# Chapter 2: Running NASM

## 2.1 NASM Command−Line Syntax

To assemble a file, you issue a command of the form

```
nasm −f <format> <filename> [−o <output>]
```

For example,

```
nasm −f elf myfile.asm
```

will assemble `myfile.asm` into an `ELF` object file `myfile.o`. And

```
nasm −f bin myfile.asm −o myfile.com
```

will assemble `myfile.asm` into a raw binary file `myfile.com`.

To produce a listing file, with the hex codes output from NASM displayed on the left of the original sources, use the `−l` option to give a listing file name, for example:

```
nasm −f coff myfile.asm −l myfile.lst
```

To get further usage instructions from NASM, try typing

```
nasm −h
```

As `−hf`, this will also list the available output file formats, and what they are.

If you use Linux but aren't sure whether your system is `a.out` or `ELF`, type

```
file nasm
```

(in the directory in which you put the NASM binary when you installed it). If it says something like

```
nasm: ELF 32−bit LSB executable i386 (386 and up) Version 1
```

then your system is `ELF`, and you should use the option `−f elf` when you want NASM to produce Linux object files. If it says

```
nasm: Linux/i386 demand−paged executable (QMAGIC)
```

or something similar, your system is `a.out`, and you should use `−f aout` instead (Linux `a.out` systems have long been obsolete, and are rare these days.)

Like Unix compilers and assemblers, NASM is silent unless it goes wrong: you won't see any output at all, unless it gives error messages.

### 2.1.1 The `−o` Option: Specifying the Output File Name

NASM will normally choose the name of your output file for you; precisely how it does this is dependent on the object file format. For Microsoft object file formats (`obj` and `win32`), it will remove the `.asm` extension (or whatever extension you like to use – NASM doesn't care) from your source file name and substitute `.obj`. For Unix object file formats (`aout`, `coff`, `elf`, `macho` and `as86`) it will substitute `.o`. For `rdf`, it will use `.rdf`, and for the `bin` format it will simply remove the extension, so that `myfile.asm` produces the output file `myfile`.

If the output file already exists, NASM will overwrite it, unless it has the same name as the input file, in which case it will give a warning and use `nasm.out` as the output file name instead.

For situations in which this behaviour is unacceptable, NASM provides the `−o` command−line option, which allows you to specify your desired output file name. You invoke `−o` by following it with the name you wish for the output file, either with or without an intervening space. For example:

```
nasm −f bin program.asm −o program.com
nasm −f bin driver.asm −odriver.sys
```

Note that this is a small o, and is different from a capital O , which is used to specify the number of optimisation passes required. See section 2.1.17.

### 2.1.2 The `−f` Option: Specifying the Output File Format

If you do not supply the `−f` option to NASM, it will choose an output file format for you itself. In the distribution versions of NASM, the default is always `bin`; if you've compiled your own copy of NASM, you can redefine `OF_DEFAULT` at compile time and choose what you want the default to be.

Like `−o`, the intervening space between `−f` and the output file format is optional; so `−f elf` and `−felf` are both valid.

A complete list of the available output file formats can be given by issuing the command `nasm −hf`.

### 2.1.3 The `−l` Option: Generating a Listing File

If you supply the `−l` option to NASM, followed (with the usual optional space) by a file name, NASM will generate a source−listing file for you, in which addresses and generated code are listed on the left, and the actual source code, with expansions of multi−line macros (except those which specifically request no expansion in source listings: see section 4.3.9) on the right. For example:

```
nasm −f elf myfile.asm −l myfile.lst
```

If a list file is selected, you may turn off listing for a section of your source with `[list −]`, and turn it back on with `[list +]`, (the default, obviously). There is no "user form" (without the brackets). This can be used to list only sections of interest, avoiding excessively long listings.

### 2.1.4 The `−M` Option: Generate Makefile Dependencies

This option can be used to generate makefile dependencies on stdout. This can be redirected to a file for further processing. For example:

```
NASM −M myfile.asm > myfile.dep
```

### 2.1.5 The `−MG` Option: Generate Makefile Dependencies

This option can be used to generate makefile dependencies on stdout. This differs from the `−M` option in that if a nonexisting file is encountered, it is assumed to be a generated file and is added to the dependency list without a prefix.

### 2.1.6 The `−F` Option: Selecting a Debug Information Format

This option is used to select the format of the debug information emitted into the output file, to be used by a debugger (or *will* be). Use of this switch does *not* enable output of the selected debug info format. Use `−g`, see section 2.1.7, to enable output.

A complete list of the available debug file formats for an output format can be seen by issuing the command `nasm −f <format> −y`. (only "borland" in "−f obj", as of 0.98.35, but "watch this space") See: section 2.1.21.

This should not be confused with the "−f dbg" output format option which is not built into NASM by default. For information on how to enable it when building from the sources, see section 6.12

### 2.1.7 The `−g` Option: Enabling Debug Information.

This option can be used to generate debugging information in the specified format. See: section 2.1.6. Using `−g` without `−F` results in emitting debug info in the default format, if any, for the selected output format. If no debug information is currently implemented in the selected output format, `−g` is *silently ignored*.

### 2.1.8 The `−X` Option: Selecting an Error Reporting Format

This option can be used to select an error reporting format for any error messages that might be produced by NASM.

Currently, two error reporting formats may be selected. They are the `−Xvc` option and the `−Xgnu` option. The GNU format is the default and looks like this:

```
filename.asm:65: error: specific error message
```

where `filename.asm` is the name of the source file in which the error was detected, `65` is the source file line number on which the error was detected, `error` is the severity of the error (this could be `warning`), and `specific error message` is a more detailed text message which should help pinpoint the exact problem.

The other format, specified by `−Xvc` is the style used by Microsoft Visual C++ and some other programs. It looks like this:

```
filename.asm(65) : error: specific error message
```

where the only difference is that the line number is in parentheses instead of being delimited by colons.

See also the `Visual C++` output format, section 6.3.

### 2.1.9 The `−Z` Option: Send Errors to a File

Under `MS−DOS` it can be difficult (though there are ways) to redirect the standard−error output of a program to a file. Since NASM usually produces its warning and error messages on `stderr`, this can make it hard to capture the errors if (for example) you want to load them into an editor.

NASM therefore provides the `−Z` option, taking a filename argument which causes errors to be sent to the specified files rather than standard error. Therefore you can redirect the errors into a file by typing

```
nasm −Z myfile.err −f obj myfile.asm
```

In earlier versions of NASM, this option was called `−E`, but it was changed since `−E` is an option conventionally used for preprocessing only, with disastrous results. See section 2.1.15.

### 2.1.10 The `−s` Option: Send Errors to `stdout`

The `−s` option redirects error messages to `stdout` rather than `stderr`, so it can be redirected under `MS−DOS`. To assemble the file `myfile.asm` and pipe its output to the `more` program, you can type:

```
nasm −s −f obj myfile.asm | more
```

See also the `−Z` option, section 2.1.9.

### 2.1.11 The `−i` Option: Include File Search Directories

When NASM sees the `%include` or `incbin` directive in a source file (see section 4.6 or section 3.2.3), it will search for the given file not only in the current directory, but also in any directories specified on the command line by the use of the `−i` option. Therefore you can include files from a macro library, for example, by typing

```
nasm −ic:\\macrolib\\ −f obj myfile.asm
```

(As usual, a space between `−i` and the path name is allowed, and optional).

NASM, in the interests of complete source−code portability, does not understand the file naming conventions of the OS it is running on; the string you provide as an argument to the `−i` option will be prepended exactly as written to the name of the include file. Therefore the trailing backslash in the above example is necessary. Under Unix, a trailing forward slash is similarly necessary.

(You can use this to your advantage, if you're really perverse, by noting that the option `−ifoo` will cause `%include "bar.i"` to search for the file `foobar.i`...)

If you want to define a *standard* include search path, similar to `/usr/include` on Unix systems, you should place one or more `−i` directives in the `NASMENV` environment variable (see section 2.1.23).

For Makefile compatibility with many C compilers, this option can also be specified as `−I`.

### 2.1.12 The `−p` Option: Pre−Include a File

NASM allows you to specify files to be *pre−included* into your source file, by the use of the `−p` option. So running

```
nasm myfile.asm −p myinc.inc
```

is equivalent to running `nasm myfile.asm` and placing the directive `%include "myinc.inc"` at the start of the file.

For consistency with the `−I`, `−D` and `−U` options, this option can also be specified as `−P`.

### 2.1.13 The `−d` Option: Pre−Define a Macro

Just as the `−p` option gives an alternative to placing `%include` directives at the start of a source file, the `−d` option gives an alternative to placing a `%define` directive. You could code

```
nasm myfile.asm −dFOO=100
```

as an alternative to placing the directive

```
%define FOO 100
```

at the start of the file. You can miss off the macro value, as well: the option `−dFOO` is equivalent to coding `%define FOO`. This form of the directive may be useful for selecting assembly−time options which are then tested using `%ifdef`, for example `−dDEBUG`.

For Makefile compatibility with many C compilers, this option can also be specified as `−D`.

### 2.1.14 The `−u` Option: Undefine a Macro

The `−u` option undefines a macro that would otherwise have been pre−defined, either automatically or by a `−p` or `−d` option specified earlier on the command lines.

For example, the following command line:

```
nasm myfile.asm −dFOO=100 −uFOO
```

would result in `FOO` *not* being a predefined macro in the program. This is useful to override options specified at a different point in a Makefile.

For Makefile compatibility with many C compilers, this option can also be specified as `−U`.

### 2.1.15 The `−E` Option: Preprocess Only

NASM allows the preprocessor to be run on its own, up to a point. Using the `−E` option (which requires no arguments) will cause NASM to preprocess its input file, expand all the macro references, remove all the comments and preprocessor directives, and print the resulting file on standard output (or save it to a file, if the `−o` option is also used).

This option cannot be applied to programs which require the preprocessor to evaluate expressions which depend on the values of symbols: so code such as

```
%assign tablesize ($−tablestart)
```

will cause an error in preprocess−only mode.

For compatiblity with older version of NASM, this option can also be written `−e`. `−E` in older versions of NASM was the equivalent of the current `−Z` option, section 2.1.9.

### 2.1.16 The `−a` Option: Don’t Preprocess At All

If NASM is being used as the back end to a compiler, it might be desirable to suppress preprocessing completely and assume the compiler has already done it, to save time and increase compilation speeds. The `−a` option, requiring no argument, instructs NASM to replace its powerful preprocessor with a stub preprocessor which does nothing.

### 2.1.17 The `−On` Option: Specifying Multipass Optimization.

NASM defaults to being a two pass assembler. This means that if you have a complex source file which needs more than 2 passes to assemble optimally, you have to enable extra passes.

Using the `−O` option, you can tell NASM to carry out multiple passes. The syntax is:

- `−O0` strict two−pass assembly, JMP and Jcc are handled more like v0.98, except that backward JMPs are short, if possible. Immediate operands take their long forms if a short form is not specified.
- `−O1` strict two−pass assembly, but forward branches are assembled with code guaranteed to reach; may produce larger code than –O0, but will produce successful assembly more often if branch offset sizes are not specified. Additionally, immediate operands which will fit in a signed byte are optimized, unless the long form is specified.
- `−On` multi−pass optimization, minimize branch offsets; also will minimize signed immediate bytes, overriding size specification unless the `strict` keyword has been used (see section 3.7). The number specifies the maximum number of passes. The more passes, the better the code, but the slower is the assembly.
- `−Ox` where `x` is the actual letter `x`, indicates to NASM to do unlimited passes.

Note that this is a capital O, and is different from a small o, which is used to specify the output format. See section 2.1.1.

### 2.1.18 The `−t` option: Enable TASM Compatibility Mode

NASM includes a limited form of compatibility with Borland’s `TASM`. When NASM’s `−t` option is used, the following changes are made:

- local labels may be prefixed with `@@` instead of `.`
- TASM−style response files beginning with `@` may be specified on the command line. This is different from the `−@resp` style that NASM natively supports.
- size override is supported within brackets. In TASM compatible mode, a size override inside square brackets changes the size of the operand, and not the address type of the operand as it

does in NASM syntax. E.g. `mov eax,[DWORD val]` is valid syntax in TASM compatibility mode. Note that you lose the ability to override the default address type for the instruction.

- `%arg` preprocessor directive is supported which is similar to TASM's `ARG` directive.
- `%local` preprocessor directive
- `%stacksize` preprocessor directive
- unprefixed forms of some directives supported (`arg`, `elif`, `else`, `endif`, `if`, `ifdef`, `ifdifi`, `ifndef`, `include`, `local`)
- more...

For more information on the directives, see the section on TASM Compatiblity preprocessor directives in section 4.9.

### 2.1.19 The `−w` Option: Enable or Disable Assembly Warnings

NASM can observe many conditions during the course of assembly which are worth mentioning to the user, but not a sufficiently severe error to justify NASM refusing to generate an output file. These conditions are reported like errors, but come up with the word 'warning' before the message. Warnings do not prevent NASM from generating an output file and returning a success status to the operating system.

Some conditions are even less severe than that: they are only sometimes worth mentioning to the user. Therefore NASM supports the `−w` command−line option, which enables or disables certain classes of assembly warning. Such warning classes are described by a name, for example `orphan−labels`; you can enable warnings of this class by the command−line option `−w+orphan−labels` and disable it by `−w−orphan−labels`.

The suppressible warning classes are:

- `macro−params` covers warnings about multi−line macros being invoked with the wrong number of parameters. This warning class is enabled by default; see section 4.3.1 for an example of why you might want to disable it.
- `macro−selfref` warns if a macro references itself. This warning class is enabled by default.
- `orphan−labels` covers warnings about source lines which contain no instruction but define a label without a trailing colon. NASM does not warn about this somewhat obscure condition by default; see section 3.1 for an example of why you might want it to.
- `number−overflow` covers warnings about numeric constants which don't fit in 32 bits (for example, it's easy to type one too many Fs and produce `0x7ffffffff` by mistake). This warning class is enabled by default.
- `gnu−elf−extensions` warns if 8−bit or 16−bit relocations are used in `−f elf` format. The GNU extensions allow this. This warning class is enabled by default.
- In addition, warning classes may be enabled or disabled across sections of source code with `[warning +warning−name]` or `[warning −warning−name]`. No "user form" (without the brackets) exists.

### 2.1.20 The `−v` Option: Display Version Info

Typing `NASM −v` will display the version of NASM which you are using, and the date on which it was compiled.

You will need the version number if you report a bug.

### 2.1.21 The `−y` Option: Display Available Debug Info Formats

Typing `nasm −f <option> −y` will display a list of the available debug info formats for the given output format. The default format is indicated by an asterisk. E.g. `nasm −f obj −y` yields `* borland`. (as of 0.98.35, the *only* debug info format implemented).

### 2.1.22 The `−−prefix` and `−−postfix` Options.

The `−−prefix` and `−−postfix` options prepend or append (respectively) the given argument to all `global` or `extern` variables. E.g. `−−prefix_` will prepend the underscore to all global and external variables, as C sometimes (but not always) likes it.

### 2.1.23 The `NASMENV` Environment Variable

If you define an environment variable called `NASMENV`, the program will interpret it as a list of extra command−line options, which are processed before the real command line. You can use this to define standard search directories for include files, by putting `−i` options in the `NASMENV` variable.

The value of the variable is split up at white space, so that the value `−s −ic:\nasmlib` will be treated as two separate options. However, that means that the value `−dNAME="my name"` won't do what you might want, because it will be split at the space and the NASM command−line processing will get confused by the two nonsensical words `−dNAME="my` and `name"`.

To get round this, NASM provides a feature whereby, if you begin the `NASMENV` environment variable with some character that isn't a minus sign, then NASM will treat this character as the separator character for options. So setting the `NASMENV` variable to the value `!−s!−ic:\nasmlib` is equivalent to setting it to `−s −ic:\nasmlib`, but `!−dNAME="my name"` will work.

This environment variable was previously called `NASM`. This was changed with version 0.98.31.

## 2.2 Quick Start for MASM Users

If you're used to writing programs with MASM, or with TASM in MASM−compatible (non−Ideal) mode, or with `a86`, this section attempts to outline the major differences between MASM's syntax and NASM's. If you're not already used to MASM, it's probably worth skipping this section.

### 2.2.1 NASM Is Case−Sensitive

One simple difference is that NASM is case−sensitive. It makes a difference whether you call your label `foo`, `Foo` or `FOO`. If you're assembling to `DOS` or `OS/2 .OBJ` files, you can invoke the `UPPERCASE` directive (documented in section 6.2) to ensure that all symbols exported to other code modules are forced to be upper case; but even then, *within* a single module, NASM will distinguish between labels differing only in case.

### 2.2.2 NASM Requires Square Brackets For Memory References

NASM was designed with simplicity of syntax in mind. One of the design goals of NASM is that it should be possible, as far as is practical, for the user to look at a single line of NASM code and tell what opcode is generated by it. You can't do this in MASM: if you declare, for example,

```
foo     equu    1
bar     dw      2
```

then the two lines of code

```
        mov     ax,foo
        mov     ax,bar
```

generate completely different opcodes, despite having identical−looking syntaxes.

NASM avoids this undesirable situation by having a much simpler syntax for memory references. The rule is simply that any access to the *contents* of a memory location requires square brackets around the address, and any access to the *address* of a variable doesn't. So an instruction of the form `mov ax,foo` will *always* refer to a compile−time constant, whether it's an `EQU` or the address of a variable; and to access the *contents* of the variable `bar`, you must code `mov ax,[bar]`.

This also means that NASM has no need for MASM's `OFFSET` keyword, since the MASM code `mov ax,offset bar` means exactly the same thing as NASM's `mov ax,bar`. If you're trying to get large amounts of MASM code to assemble sensibly under NASM, you can always code `%idefine offset` to make the preprocessor treat the `OFFSET` keyword as a no−op.

This issue is even more confusing in `a86`, where declaring a label with a trailing colon defines it to be a 'label' as opposed to a 'variable' and causes `a86` to adopt NASM−style semantics; so in `a86`, `mov ax,var` has different behaviour depending on whether `var` was declared as `var: dw 0` (a label) or `var dw 0` (a word−size variable). NASM is very simple by comparison: *everything* is a label.

NASM, in the interests of simplicity, also does not support the hybrid syntaxes supported by MASM and its clones, such as `mov ax,table[bx]`, where a memory reference is denoted by one portion outside square brackets and another portion inside. The correct syntax for the above is `mov ax,[table+bx]`. Likewise, `mov ax,es:[di]` is wrong and `mov ax,[es:di]` is right.

### 2.2.3 NASM Doesn't Store Variable Types

NASM, by design, chooses not to remember the types of variables you declare. Whereas MASM will remember, on seeing `var dw 0`, that you declared `var` as a word−size variable, and will then be able to fill in the ambiguity in the size of the instruction `mov var,2`, NASM will deliberately remember nothing about the symbol `var` except where it begins, and so you must explicitly code `mov word [var],2`.

For this reason, NASM doesn't support the `LODS`, `MOVS`, `STOS`, `SCAS`, `CMPS`, `INS`, or `OUTS` instructions, but only supports the forms such as `LODSB`, `MOVSW`, and `SCASD`, which explicitly specify the size of the components of the strings being manipulated.

### 2.2.4 NASM Doesn't `ASSUME`

As part of NASM's drive for simplicity, it also does not support the `ASSUME` directive. NASM will not keep track of what values you choose to put in your segment registers, and will never *automatically* generate a segment override prefix.

### 2.2.5 NASM Doesn't Support Memory Models

NASM also does not have any directives to support different 16−bit memory models. The programmer has to keep track of which functions are supposed to be called with a far call and which with a near call, and is responsible for putting the correct form of `RET` instruction (`RETN` or `RETF`; NASM accepts `RET` itself as an alternate form for `RETN`); in addition, the programmer is responsible for coding CALL FAR instructions where necessary when calling *external* functions, and must also keep track of which external variable definitions are far and which are near.

### 2.2.6 Floating−Point Differences

NASM uses different names to refer to floating−point registers from MASM: where MASM would call them `ST(0)`, `ST(1)` and so on, and `a86` would call them simply `0`, `1` and so on, NASM chooses to call them `st0`, `st1` etc.

As of version 0.96, NASM now treats the instructions with ‘nowait’ forms in the same way as MASM−compatible assemblers. The idiosyncratic treatment employed by 0.95 and earlier was based on a misunderstanding by the authors.

### 2.2.7 Other Differences

For historical reasons, NASM uses the keyword `TWORD` where MASM and compatible assemblers use `TBYTE`.

NASM does not declare uninitialized storage in the same way as MASM: where a MASM programmer might use `stack db 64 dup (?)`, NASM requires `stack resb 64`, intended to be read as ‘reserve 64 bytes’. For a limited amount of compatibility, since NASM treats `?` as a valid character in symbol names, you can code `? equ 0` and then writing `dw ?` will at least do something vaguely useful. `DUP` is still not a supported syntax, however.

In addition to all of this, macros and directives work completely differently to MASM. See chapter 4 and chapter 5 for further details.

# Chapter 3: The NASM Language

## 3.1 Layout of a NASM Source Line

Like most assemblers, each NASM source line contains (unless it is a macro, a preprocessor directive or an assembler directive: see chapter 4 and chapter 5) some combination of the four fields

```
label:    instruction operands        ; comment
```

As usual, most of these fields are optional; the presence or absence of any combination of a label, an instruction and a comment is allowed. Of course, the operand field is either required or forbidden by the presence and nature of the instruction field.

NASM uses backslash (\) as the line continuation character; if a line ends with backslash, the next line is considered to be a part of the backslash−ended line.

NASM places no restrictions on white space within a line: labels may have white space before them, or instructions may have no space before them, or anything. The colon after a label is also optional. (Note that this means that if you intend to code `lodsb` alone on a line, and type `lodab` by accident, then that's still a valid source line which does nothing but define a label. Running NASM with the command−line option `−w+orphan−labels` will cause it to warn you if you define a label alone on a line without a trailing colon.)

Valid characters in labels are letters, numbers, `_`, `$`, `#`, `@`, `~`, `.`, and `?`. The only characters which may be used as the *first* character of an identifier are letters, `.` (with special meaning: see section 3.9), `_` and `?`. An identifier may also be prefixed with a `$` to indicate that it is intended to be read as an identifier and not a reserved word; thus, if some other module you are linking with defines a symbol called `eax`, you can refer to `$eax` in NASM code to distinguish the symbol from the register. Maximum length of an identifier is 4095 characters.

The instruction field may contain any machine instruction: Pentium and P6 instructions, FPU instructions, MMX instructions and even undocumented instructions are all supported. The instruction may be prefixed by `LOCK`, `REP`, `REPE`/`REPZ` or `REPNE`/`REPNZ`, in the usual way. Explicit address−size and operand−size prefixes `A16`, `A32`, `O16` and `O32` are provided – one example of their use is given in chapter 9. You can also use the name of a segment register as an instruction prefix: coding `es mov [bx],ax` is equivalent to coding `mov [es:bx],ax`. We recommend the latter syntax, since it is consistent with other syntactic features of the language, but for instructions such as `LODSB`, which has no operands and yet can require a segment override, there is no clean syntactic way to proceed apart from `es lodsb`.

An instruction is not required to use a prefix: prefixes such as `CS`, `A32`, `LOCK` or `REPE` can appear on a line by themselves, and NASM will just generate the prefix bytes.

In addition to actual machine instructions, NASM also supports a number of pseudo−instructions, described in section 3.2.

Instruction operands may take a number of forms: they can be registers, described simply by the register name (e.g. `ax`, `bp`, `ebx`, `cr0`: NASM does not use the `gas`–style syntax in which register names must be prefixed by a `%` sign), or they can be effective addresses (see section 3.3), constants (section 3.4) or expressions (section 3.5).

For x87 floating−point instructions, NASM accepts a wide range of syntaxes: you can use two−operand forms like MASM supports, or you can use NASM's native single−operand forms in most cases. For example, you can code:

```
fadd    st1             ; this sets st0 := st0 + st1
fadd    st0,st1         ; so does this

fadd    st1,st0         ; this sets st1 := st1 + st0
fadd    to st1          ; so does this
```

Almost any x87 floating−point instruction that references memory must use one of the prefixes `DWORD`, `QWORD` or `TWORD` to indicate what size of memory operand it refers to.

## 3.2 Pseudo−Instructions

Pseudo−instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them. The current pseudo−instructions are `DB`, `DW`, `DD`, `DQ`, `DT` and `DO`; their uninitialized counterparts `RESB`, `RESW`, `RESD`, `RESQ`, `REST` and `RESO`; the `INCBIN` command, the `EQU` command, and the `TIMES` prefix.

### 3.2.1 `DB` and friends: Declaring initialized Data

`DB`, `DW`, `DD`, `DQ`, `DT` and `DO` are used, much as in MASM, to declare initialized data in the output file. They can be invoked in a wide range of ways:

```
db    0x55                ; just the byte 0x55
db    0x55,0x56,0x57      ; three bytes in succession
db    'a',0x55            ; character constants are OK
db    'hello',13,10,'$'   ; so are string constants
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x61 0x00 (it's just a number)
dw    'ab'                ; 0x61 0x62 (character constant)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; floating−point constant
dq    0x123456789abcdef0  ; eight byte constant
dq    1.234567e20         ; double−precision float
dt    1.234567e20         ; extended−precision float
```

`DT` and `DO` do not accept numeric constants as operands. `DB` does not accept floating−point numbers as operands.

### 3.2.2 `RESB` and friends: Declaring Uninitialized Data

`RESB`, `RESW`, `RESD`, `RESQ`, `REST` and `RESO` are designed to be used in the BSS section of a module: they declare *uninitialized* storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve. As stated in section 2.2.7, NASM does not support the MASM/TASM syntax of reserving uninitialized space by writing `DW ?` or similar things: this is what it does instead. The operand to a `RESB`–type pseudo−instruction is a *critical expression*: see section 3.8.

For example:

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray       resq    10              ; array of ten reals
```

### 3.2.3 `INCBIN`: Including External Binary Files

`INCBIN` is borrowed from the old Amiga assembler DevPac: it includes a binary file verbatim into the output file. This can be handy for (for example) including graphics and sound data directly into a game executable file. It can be called in one of these three ways:

```
    incbin  "file.dat"             ; include the whole file
    incbin  "file.dat",1024        ; skip the first 1024 bytes
    incbin  "file.dat",1024,512    ; skip the first 1024, and
                                   ; actually include at most 512
```

### 3.2.4 `EQU`: Defining Constants

`EQU` defines a symbol to a given constant value: when `EQU` is used, the source line must contain a label. The action of `EQU` is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. So, for example,

```
message         db      'hello, world'
msglen          equ     $-message
```

defines `msglen` to be the constant 12. `msglen` may not then be redefined later. This is not a preprocessor definition either: the value of `msglen` is evaluated *once*, using the value of `$` (see section 3.5 for an explanation of `$`) at the point of definition, rather than being evaluated wherever it is referenced and using the value of `$` at the point of reference. Note that the operand to an `EQU` is also a critical expression (section 3.8).

### 3.2.5 `TIMES`: Repeating Instructions or Data

The `TIMES` prefix causes the instruction to be assembled multiple times. This is partly present as NASM's equivalent of the `DUP` syntax supported by MASM–compatible assemblers, in that you can code

```
zerobuf:        times 64 db 0
```

or similar things; but `TIMES` is more versatile than that. The argument to `TIMES` is not just a numeric constant, but a numeric *expression*, so you can do things like

```
buffer: db      'hello, world'
        times 64-$+buffer db ' '
```

which will store exactly enough spaces to make the total length of `buffer` up to 64. Finally, `TIMES` can be applied to ordinary instructions, so you can code trivial unrolled loops in it:

```
        times 100 movsb
```

Note that there is no effective difference between `times 100 resb 1` and `resb 100`, except that the latter will be assembled about 100 times faster due to the internal structure of the assembler.

The operand to `TIMES`, like that of `EQU` and those of `RESB` and friends, is a critical expression (section 3.8).

Note also that `TIMES` can't be applied to macros: the reason for this is that `TIMES` is processed after the macro phase, which allows the argument to `TIMES` to contain expressions such as `64-$+buffer` as above. To repeat more than one line of code, or a complex macro, use the preprocessor `%rep` directive.

## 3.3 Effective Addresses

An effective address is any operand to an instruction which references memory. Effective addresses, in NASM, have a very simple syntax: they consist of an expression evaluating to the desired address, enclosed in square brackets. For example:

```
wordvar dw      123
        mov     ax,[wordvar]
        mov     ax,[wordvar+1]
        mov     ax,[es:wordvar+bx]
```

Anything not conforming to this simple system is not a valid memory reference in NASM, for example `es:wordvar[bx]`.

More complicated effective addresses, such as those involving more than one register, work in exactly the same way:

```
        mov     eax,[ebx*2+ecx+offset]
        mov     ax,[bp+di+8]
```

NASM is capable of doing algebra on these effective addresses, so that things which don't necessarily *look* legal are perfectly all right:

```
    mov     eax,[ebx*5]             ; assembles as [ebx*4+ebx]
    mov     eax,[label1*2-label2]   ; ie [label1+(label1-label2)]
```

Some forms of effective address have more than one assembled form; in most such cases NASM will generate the smallest form it can. For example, there are distinct assembled forms for the 32−bit effective addresses `[eax*2+0]` and `[eax+eax]`, and NASM will generally generate the latter on the grounds that the former requires four bytes to store a zero offset.

NASM has a hinting mechanism which will cause `[eax+ebx]` and `[ebx+eax]` to generate different opcodes; this is occasionally useful because `[esi+ebp]` and `[ebp+esi]` have different default segment registers.

However, you can force NASM to generate an effective address in a particular form by the use of the keywords `BYTE`, `WORD`, `DWORD` and `NOSPLIT`. If you need `[eax+3]` to be assembled using a double−word offset field instead of the one byte NASM will normally generate, you can code `[dword eax+3]`. Similarly, you can force NASM to use a byte offset for a small value which it hasn't seen on the first pass (see section 3.8 for an example of such a code fragment) by using `[byte eax+offset]`. As special cases, `[byte eax]` will code `[eax+0]` with a byte offset of zero, and `[dword eax]` will code it with a double−word offset of zero. The normal form, `[eax]`, will be coded with no offset field.

The form described in the previous paragraph is also useful if you are trying to access data in a 32−bit segment from within 16 bit code. For more information on this see the section on mixed−size addressing (section 9.2). In particular, if you need to access data with a known offset that is larger than will fit in a 16−bit value, if you don't specify that it is a dword offset, nasm will cause the high word of the offset to be lost.

Similarly, NASM will split `[eax*2]` into `[eax+eax]` because that allows the offset field to be absent and space to be saved; in fact, it will also split `[eax*2+offset]` into `[eax+eax+offset]`. You can combat this behaviour by the use of the `NOSPLIT` keyword: `[nosplit eax*2]` will force `[eax*2+0]` to be generated literally.

In 64−bit mode, NASM will by default generate absolute addresses. The `REL` keyword makes it produce `RIP`–relative addresses. Since this is frequently the normally desired behaviour, see the `DEFAULT` directive (section 5.2). The keyword `ABS` overrides `REL`.

## 3.4 Constants

NASM understands four different types of constant: numeric, character, string and floating−point.

### 3.4.1 Numeric Constants

A numeric constant is simply a number. NASM allows you to specify numbers in a variety of number bases, in a variety of ways: you can suffix `H`, `Q` or `O`, and `B` for hex, octal and binary, or you can prefix `0x` for hex in the style of C, or you can prefix `$` for hex in the style of Borland Pascal. Note, though, that the `$` prefix does double duty as a prefix on identifiers (see section 3.1), so a hex number prefixed with a `$` sign must have a digit after the `$` rather than a letter.

Some examples:

```
        mov     ax,100          ; decimal
        mov     ax,0a2h         ; hex
        mov     ax,$0a2         ; hex again: the 0 is required
        mov     ax,0xa2         ; hex yet again
        mov     ax,777q         ; octal
        mov     ax,777o         ; octal again
        mov     ax,10010011b    ; binary
```

### 3.4.2 Character Constants

A character constant consists of up to four characters enclosed in either single or double quotes. The type of quote makes no difference to NASM, except of course that surrounding the constant with single quotes allows double quotes to appear within it and vice versa.

A character constant with more than one character will be arranged with little−endian order in mind: if you code

```
          mov eax,'abcd'
```

then the constant generated is not `0x61626364`, but `0x64636261`, so that if you were then to store the value into memory, it would read `abcd` rather than `dcba`. This is also the sense of character constants understood by the Pentium's `CPUID` instruction.

### 3.4.3 String Constants

String constants are only acceptable to some pseudo−instructions, namely the `DB` family and `INCBIN`.

A string constant looks like a character constant, only longer. It is treated as a concatenation of maximum−size character constants for the conditions. So the following are equivalent:

```
      db    'hello'               ; string constant
      db    'h','e','l','l','o'   ; equivalent character constants
```

And the following are also equivalent:

```
      dd    'ninechars'           ; doubleword string constant
      dd    'nine','char','s'     ; becomes three doublewords
      db    'ninechars',0,0,0     ; and really looks like this
```

Note that when used as an operand to `db`, a constant like `'ab'` is treated as a string constant despite being short enough to be a character constant, because otherwise `db 'ab'` would have the same effect as `db 'a'`, which would be silly. Similarly, three−character or four−character constants are treated as strings when they are operands to `dw`.

### 3.4.4 Floating−Point Constants

Floating−point constants are acceptable only as arguments to `DW`, `DD`, `DQ`, `DT`, and `DO`, or as arguments to the special operators `__float16__`, `__float32__`, `__float64__`, `__float80m__`, `__float80e__`, `__float128l__`, and `__float128h__`.

Floating−point constants are expressed in the traditional form: digits, then a period, then optionally more digits, then optionally an `E` followed by an exponent. The period is mandatory, so that NASM can distinguish between `dd 1`, which declares an integer constant, and `dd 1.0` which declares a floating−point constant. NASM also support C99−style hexadecimal floating−point: `0x`, hexadecimal digits, period, optionally more hexadeximal digits, then optionally a `P` followed by a *binary* (not hexadecimal) exponent in decimal notation.

Some examples:

```
      dw    -0.5                    ; IEEE half precision
      dd    1.2                     ; an easy one
```

The special operators are used to produce floating−point numbers in other contexts. They produce the binary representation of a specific floating−point number as an integer, and can use anywhere integer constants are used in an expression. `__float80m__` and `__float80e__` produce the 64−bit mantissa and 16−bit exponent of an 80−bit floating−point number, and `__float128l__` and `__float128h__` produce the lower and upper 64−bit halves of a 128−bit floating−point number, respectively.

For example:

```
      mov    rax,__float64__(3.141592653589793238462)
```

... would assign the binary representation of pi as a 64−bit floating point number into `RAX`. This is exactly equivalent to:

```
      mov    rax,0x401921fb54442d18
```

NASM cannot do compile−time arithmetic on floating−point constants. This is because NASM is designed to be portable – although it always generates code to run on x86 processors, the assembler itself can run on any system with an ANSI C compiler. Therefore, the assembler cannot guarantee the presence of a floating−point unit capable of handling the Intel number formats, and so for NASM to be able to do floating arithmetic it would have to include its own complete set of floating−point routines, which would significantly increase the size of the assembler for very little benefit.

The special tokens `__Infinity__`, `__QNaN__` (or `__NaN__`) and `__SNaN__` can be used to generate infinities, quiet NaNs, and signalling NaNs, respectively. These are normally used as macros:

```
%define Inf __Infinity__
%define NaN __QNaN__

      dq    +1.5, -Inf, NaN         ; Double-precision constants
```

## 3.5 Expressions

Expressions in NASM are similar in syntax to those in C. Expressions are evaluated as 64−bit integers which are then adjusted to the appropriate size.

NASM supports two special tokens in expressions, allowing calculations to involve the current assembly position: the `$` and `$$` tokens. `$` evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using `JMP $`. `$$` evaluates to the beginning of the current section; so you can tell how far into the section you are by using `($-$$)`.

The arithmetic operators provided by NASM are listed here, in increasing order of precedence.

### 3.5.1 `|`: Bitwise OR Operator

The `|` operator gives a bitwise OR, exactly as performed by the `OR` machine instruction. Bitwise OR is the lowest−priority arithmetic operator supported by NASM.

### 3.5.2 `^`: Bitwise XOR Operator

`^` provides the bitwise XOR operation.

### 3.5.3 `&`: Bitwise AND Operator

`&` provides the bitwise AND operation.

### 3.5.4 `<<` and `>>`: Bit Shift Operators

`<<` gives a bit−shift to the left, just as it does in C. So `5<<3` evaluates to 5 times 8, or 40. `>>` gives a bit−shift to the right; in NASM, such a shift is *always* unsigned, so that the bits shifted in from the left−hand end are filled with zero rather than a sign−extension of the previous highest bit.

### 3.5.5 `+` and `−`: Addition and Subtraction Operators

The `+` and `−` operators do perfectly ordinary addition and subtraction.

### 3.5.6 `*`, `/`, `//`, `%` and `%%`: Multiplication and Division

`*` is the multiplication operator. `/` and `//` are both division operators: `/` is unsigned division and `//` is signed division. Similarly, `%` and `%%` provide unsigned and signed modulo operators respectively.

NASM, like ANSI C, provides no guarantees about the sensible operation of the signed modulo operator.

Since the `%` character is used extensively by the macro preprocessor, you should ensure that both the signed and unsigned modulo operators are followed by white space wherever they appear.

### 3.5.7 Unary Operators: `+`, `−`, `~`, `!` and `SEG`

The highest−priority operators in NASM's expression grammar are those which only apply to one argument. `−` negates its operand, `+` does nothing (it's provided for symmetry with `−`), `~` computes the one's complement of its operand, `!` is the logical negation operator, and `SEG` provides the segment address of its operand (explained in more detail in section 3.6).

## 3.6 `SEG` and `WRT`

When writing large 16−bit programs, which must be split into multiple segments, it is often necessary to be able to refer to the segment part of the address of a symbol. NASM supports the `SEG` operator to perform this function.

The `SEG` operator returns the *preferred* segment base of a symbol, defined as the segment base relative to which the offset of the symbol makes sense. So the code

```
        mov     ax,seg symbol
        mov     es,ax
        mov     bx,symbol
```

will load `ES:BX` with a valid pointer to the symbol `symbol`.

Things can be more complex than this: since 16−bit segments and groups may overlap, you might occasionally want to refer to some symbol using a different segment base from the preferred one. NASM lets you do this, by the use of the `WRT` (With Reference To) keyword. So you can do things like

```
        mov     ax,weird_seg        ; weird_seg is a segment base
        mov     es,ax
        mov     bx,symbol wrt weird_seg
```

to load `ES:BX` with a different, but functionally equivalent, pointer to the symbol `symbol`.

NASM supports far (inter−segment) calls and jumps by means of the syntax `call segment:offset`, where `segment` and `offset` both represent immediate values. So to call a far procedure, you could code either of

```
        call    (seg procedure):procedure
        call    weird_seg:(procedure wrt weird_seg)
```

(The parentheses are included for clarity, to show the intended parsing of the above instructions. They are not necessary in practice.)

NASM supports the syntax `call far procedure` as a synonym for the first of the above usages. `JMP` works identically to `CALL` in these examples.

To declare a far pointer to a data item in a data segment, you must code

```
        dw      symbol, seg symbol
```

NASM supports no convenient synonym for this, though you can always invent one using the macro processor.

## 3.7 `STRICT`: Inhibiting Optimization

When assembling with the optimizer set to level 2 or higher (see section 2.1.17), NASM will use size specifiers (`BYTE`, `WORD`, `DWORD`, `QWORD`, `TWORD` or `OWORD`), but will give them the smallest possible size. The keyword `STRICT` can be used to inhibit optimization and force a particular operand to be emitted in the specified size. For example, with the optimizer on, and in `BITS 16` mode,

```
        push dword 33
```

is encoded in three bytes `66 6A 21`, whereas

```
        push strict dword 33
```

is encoded in six bytes, with a full dword immediate operand `66 68 21 00 00 00`.

With the optimizer off, the same code (six bytes) is generated whether the `STRICT` keyword was used or not.

## 3.8 Critical Expressions

A limitation of NASM is that it is a two−pass assembler; unlike TASM and others, it will always do exactly two assembly passes. Therefore it is unable to cope with source files that are complex enough to require three or more passes.

The first pass is used to determine the size of all the assembled code and data, so that the second pass, when generating all the code, knows all the symbol addresses the code refers to. So one thing NASM can’t handle is code whose size depends on the value of a symbol declared after the code in question. For example,

```
        times (label−$) db 0
label:  db      ’Where am I?’
```

The argument to `TIMES` in this case could equally legally evaluate to anything at all; NASM will reject this example because it cannot tell the size of the `TIMES` line when it first sees it. It will just as firmly reject the slightly paradoxical code

```
        times (label−$+1) db 0
label:  db      ’NOW where am I?’
```

in which *any* value for the `TIMES` argument is by definition wrong!

NASM rejects these examples by means of a concept called a *critical expression*, which is defined to be an expression whose value is required to be computable in the first pass, and which must therefore depend only on symbols defined before it. The argument to the `TIMES` prefix is a critical expression; for the same reason, the arguments to the `RESB` family of pseudo−instructions are also critical expressions.

Critical expressions can crop up in other contexts as well: consider the following code.

```
                mov     ax,symbol1
symbol1         equ     symbol2
symbol2:
```

On the first pass, NASM cannot determine the value of `symbol1`, because `symbol1` is defined to be equal to `symbol2` which NASM hasn't seen yet. On the second pass, therefore, when it encounters the line `mov ax,symbol1`, it is unable to generate the code for it because it still doesn't know the value of `symbol1`. On the next line, it would see the `EQU` again and be able to determine the value of `symbol1`, but by then it would be too late.

NASM avoids this problem by defining the right−hand side of an `EQU` statement to be a critical expression, so the definition of `symbol1` would be rejected in the first pass.

There is a related issue involving forward references: consider this code fragment.

```
        mov     eax,[ebx+offset]
offset  equ     10
```

NASM, on pass one, must calculate the size of the instruction `mov eax,[ebx+offset]` without knowing the value of `offset`. It has no way of knowing that `offset` is small enough to fit into a one−byte offset field and that it could therefore get away with generating a shorter form of the effective−address encoding; for all it knows, in pass one, `offset` could be a symbol in the code segment, and it might need the full four−byte form. So it is forced to compute the size of the instruction to accommodate a four−byte address part. In pass two, having made this decision, it is now forced to honour it and keep the instruction large, so the code generated in this case is not as small as it could have been. This problem can be solved by defining `offset` before using it, or by forcing byte size in the effective address by coding `[byte ebx+offset]`.

Note that use of the `−On` switch (with n>=2) makes some of the above no longer true (see section 2.1.17).

## 3.9 Local Labels

NASM gives special treatment to symbols beginning with a period. A label beginning with a single period is treated as a *local* label, which means that it is associated with the previous non−local label. So, for example:

```
label1  ; some code

.loop
        ; some more code

        jne     .loop
        ret

label2  ; some code

.loop
        ; some more code

        jne     .loop
        ret
```

In the above code fragment, each `JNE` instruction jumps to the line immediately before it, because the two definitions of `.loop` are kept separate by virtue of each being associated with the previous non−local label.

This form of local label handling is borrowed from the old Amiga assembler DevPac; however, NASM goes one step further, in allowing access to local labels from other parts of the code. This is achieved by means of *defining* a local label in terms of the previous non−local label: the first definition of `.loop` above is really defining a symbol called `label1.loop`, and the second defines a symbol called `label2.loop`. So, if you really needed to, you could write

```
label3  ; some more code
        ; and some more

        jmp label1.loop
```

Sometimes it is useful – in a macro, for instance – to be able to define a label which can be referenced from anywhere but which doesn't interfere with the normal local−label mechanism. Such a label can't be non−local because it would interfere with subsequent definitions of, and references to, local labels; and it can't be local because the macro that defined it wouldn't know the label's full name. NASM therefore introduces a third type of label, which is probably only useful in macro definitions: if a label begins with the special prefix `..@`, then it does nothing to the local label mechanism. So you could code

```
label1:                         ; a non−local label
.local:                         ; this is really label1.local
..@@foo:                         ; this is a special symbol
label2:                         ; another non−local label
.local:                         ; this is really label2.local

        jmp     ..@@foo          ; this will jump three lines up
```

NASM has the capacity to define other special symbols beginning with a double period: for example, `..start` is used to specify the entry point in the `obj` output format (see section 6.2.6).

# Chapter 4: The NASM Preprocessor

NASM contains a powerful macro processor, which supports conditional assembly, multi−level file inclusion, two forms of macro (single−line and multi−line), and a ‘context stack’ mechanism for extra macro power. Preprocessor directives all begin with a `%` sign.

The preprocessor collapses all lines which end with a backslash (\) character into a single line. Thus:

```
%define THIS_VERY_LONG_MACRO_NAME_IS_DEFINED_TO \\
        THIS_VALUE
```

will work like a single−line macro without the backslash−newline sequence.

## 4.1 Single−Line Macros

### 4.1.1 The Normal Way: `%define`

Single−line macros are defined using the `%define` preprocessor directive. The definitions work in a similar way to C; so you can do things like

```
%define ctrl    0x1F &
%define param(a,b) ((a)+(a)*(b))

        mov     byte [param(2,ebx)], ctrl ’D’
```

which will expand to

```
        mov     byte [(2)+(2)*(ebx)], 0x1F & ’D’
```

When the expansion of a single−line macro contains tokens which invoke another macro, the expansion is performed at invocation time, not at definition time. Thus the code

```
%define a(x)    1+b(x)
%define b(x)    2*x

        mov     ax,a(8)
```

will evaluate in the expected way to `mov ax,1+2*8`, even though the macro `b` wasn’t defined at the time of definition of `a`.

Macros defined with `%define` are case sensitive: after `%define foo bar`, only `foo` will expand to `bar`: `Foo` or `FOO` will not. By using `%idefine` instead of `%define` (the ‘i’ stands for ‘insensitive’) you can define all the case variants of a macro at once, so that `%idefine foo bar` would cause `foo`, `Foo`, `FOO`, `fOO` and so on all to expand to `bar`.

There is a mechanism which detects when a macro call has occurred as a result of a previous expansion of the same macro, to guard against circular references and infinite loops. If this happens, the preprocessor will only expand the first occurrence of the macro. Hence, if you code

```
%define a(x)    1+a(x)

        mov     ax,a(3)
```

the macro `a(3)` will expand once, becoming `1+a(3)`, and will then expand no further. This behaviour can be useful: see section 8.1 for an example of its use.

You can overload single−line macros: if you write

```
%define foo(x)   1+x
%define foo(x,y) 1+x*y
```

the preprocessor will be able to handle both types of macro call, by counting the parameters you pass; so `foo(3)` will become `1+3` whereas `foo(ebx,2)` will become `1+ebx*2`. However, if you define

```
%define foo bar
```

then no other definition of `foo` will be accepted: a macro with no parameters prohibits the definition of the same name as a macro *with* parameters, and vice versa.

This doesn't prevent single−line macros being *redefined*: you can perfectly well define a macro with

```
%define foo bar
```

and then re−define it later in the same source file with

```
%define foo baz
```

Then everywhere the macro `foo` is invoked, it will be expanded according to the most recent definition. This is particularly useful when defining single−line macros with `%assign` (see section 4.1.5).

You can pre−define single−line macros using the '−d' option on the NASM command line: see section 2.1.13.

### 4.1.2 Enhancing %define: `%xdefine`

To have a reference to an embedded single−line macro resolved at the time that it is embedded, as opposed to when the calling macro is expanded, you need a different mechanism to the one offered by `%define`. The solution is to use `%xdefine`, or it's case−insensitive counterpart `%xidefine`.

Suppose you have the following code:

```
%define  isTrue  1
%define  isFalse isTrue
%define  isTrue  0

val1:    db      isFalse

%define  isTrue  1

val2:    db      isFalse
```

In this case, `val1` is equal to 0, and `val2` is equal to 1. This is because, when a single−line macro is defined using `%define`, it is expanded only when it is called. As `isFalse` expands to `isTrue`, the expansion will be the current value of `isTrue`. The first time it is called that is 0, and the second time it is 1.

If you wanted `isFalse` to expand to the value assigned to the embedded macro `isTrue` at the time that `isFalse` was defined, you need to change the above code to use `%xdefine`.

```
%xdefine isTrue  1
%xdefine isFalse isTrue
%xdefine isTrue  0

val1:    db      isFalse
```

```
%xdefine isTrue  1

val2:    db      isFalse
```

Now, each time that `isFalse` is called, it expands to 1, as that is what the embedded macro `isTrue` expanded to at the time that `isFalse` was defined.

### 4.1.3 Concatenating Single Line Macro Tokens: `%+`

Individual tokens in single line macros can be concatenated, to produce longer tokens for later processing. This can be useful if there are several similar macros that perform similar functions.

Please note that a space is required after `%+`, in order to disambiguate it from the syntax `%+1` used in multiline macros.

As an example, consider the following:

```
%define BDASTART 400h                ; Start of BIOS data area

struc   tBIOSDA                      ; its structure
        .COM1addr       RESW    1
        .COM2addr       RESW    1
        ; ..and so on
endstruc
```

Now, if we need to access the elements of tBIOSDA in different places, we can end up with:

```
        mov     ax,BDASTART + tBIOSDA.COM1addr
        mov     bx,BDASTART + tBIOSDA.COM2addr
```

This will become pretty ugly (and tedious) if used in many places, and can be reduced in size significantly by using the following macro:

```
; Macro to access BIOS variables by their names (from tBDA):

%define BDA(x)  BDASTART + tBIOSDA. %+ x
```

Now the above code can be written as:

```
        mov     ax,BDA(COM1addr)
        mov     bx,BDA(COM2addr)
```

Using this feature, we can simplify references to a lot of macros (and, in turn, reduce typing errors).

### 4.1.4 Undefining macros: `%undef`

Single−line macros can be removed with the `%undef` command. For example, the following sequence:

```
%define foo bar
%undef  foo

        mov     eax, foo
```

will expand to the instruction `mov eax, foo`, since after `%undef` the macro `foo` is no longer defined.

Macros that would otherwise be pre−defined can be undefined on the command−line using the '−u' option on the NASM command line: see section 2.1.14.

### 4.1.5 Preprocessor Variables: `%assign`

An alternative way to define single−line macros is by means of the `%assign` command (and its case−insensitive counterpart `%iassign`, which differs from `%assign` in exactly the same way that `%idefine` differs from `%define`).

`%assign` is used to define single−line macros which take no parameters and have a numeric value. This value can be specified in the form of an expression, and it will be evaluated once, when the `%assign` directive is processed.

Like `%define`, macros defined using `%assign` can be re−defined later, so you can do things like

```
%assign i i+1
```

to increment the numeric value of a macro.

`%assign` is useful for controlling the termination of `%rep` preprocessor loops: see section 4.5 for an example of this. Another use for `%assign` is given in section 7.4 and section 8.1.

The expression passed to `%assign` is a critical expression (see section 3.8), and must also evaluate to a pure number (rather than a relocatable reference such as a code or data address, or anything involving a register).

## 4.2 String Handling in Macros: `%strlen` and `%substr`

It’s often useful to be able to handle strings in macros. NASM supports two simple string handling macro operators from which more complex operations can be constructed.

### 4.2.1 String Length: `%strlen`

The `%strlen` macro is like `%assign` macro in that it creates (or redefines) a numeric value to a macro. The difference is that with `%strlen`, the numeric value is the length of a string. An example of the use of this would be:

```
%strlen charcnt ’my string’
```

In this example, `charcnt` would receive the value 8, just as if an `%assign` had been used. In this example, `’my string’` was a literal string but it could also have been a single−line macro that expands to a string, as in the following example:

```
%define sometext ’my string’
%strlen charcnt sometext
```

As in the first case, this would result in `charcnt` being assigned the value of 8.

### 4.2.2 Sub−strings: `%substr`

Individual letters in strings can be extracted using `%substr`. An example of its use is probably more useful than the description:

```
%substr mychar  ’xyz’ 1         ; equivalent to %define mychar ’x’
%substr mychar  ’xyz’ 2         ; equivalent to %define mychar ’y’
%substr mychar  ’xyz’ 3         ; equivalent to %define mychar ’z’
```

In this example, mychar gets the value of ’y’. As with `%strlen` (see section 4.2.1), the first parameter is the single−line macro to be created and the second is the string. The third parameter specifies which character is to be selected. Note that the first index is 1, not 0 and the last index is equal to the value that `%strlen` would assign given the same string. Index values out of range result in an empty string.

## 4.3 Multi−Line Macros: `%macro`

Multi−line macros are much more like the type of macro seen in MASM and TASM: a multi−line macro definition in NASM looks something like this.

```
%macro  prologue 1

        push    ebp
        mov     ebp,esp
        sub     esp,%1

%endmacro
```

This defines a C−like function prologue as a macro: so you would invoke the macro with a call such as

```
myfunc:   prologue 12
```

which would expand to the three lines of code

```
myfunc: push    ebp
        mov     ebp,esp
        sub     esp,12
```

The number `1` after the macro name in the `%macro` line defines the number of parameters the macro `prologue` expects to receive. The use of `%1` inside the macro definition refers to the first parameter to the macro call. With a macro taking more than one parameter, subsequent parameters would be referred to as `%2`, `%3` and so on.

Multi−line macros, like single−line macros, are case−sensitive, unless you define them using the alternative directive `%imacro`.

If you need to pass a comma as *part* of a parameter to a multi−line macro, you can do that by enclosing the entire parameter in braces. So you could code things like

```
%macro  silly 2

    %2: db      %1

%endmacro

        silly ’a’, letter_a             ; letter_a:  db ’a’
        silly ’ab’, string_ab           ; string_ab: db ’ab’
        silly @\{13,10@\}, crlf             ; crlf:      db 13,10
```

### 4.3.1 Overloading Multi−Line Macros

As with single−line macros, multi−line macros can be overloaded by defining the same macro name several times with different numbers of parameters. This time, no exception is made for macros with no parameters at all. So you could define

```
%macro  prologue 0

        push    ebp
        mov     ebp,esp

%endmacro
```

to define an alternative form of the function prologue which allocates no local stack space.

Sometimes, however, you might want to ‘overload’ a machine instruction; for example, you might want to define

```
%macro  push 2

        push    %1
        push    %2

%endmacro
```

so that you could code

```
        push    ebx             ; this line is not a macro call
        push    eax,ecx         ; but this one is
```

Ordinarily, NASM will give a warning for the first of the above two lines, since `push` is now defined to be a macro, and is being invoked with a number of parameters for which no definition has been given. The correct code will still be generated, but the assembler will give a warning. This warning can be disabled by the use of the `−w−macro−params` command−line option (see section 2.1.19).

### 4.3.2 Macro−Local Labels

NASM allows you to define labels within a multi−line macro definition in such a way as to make them local to the macro call: so calling the same macro multiple times will use a different label each time. You do this by prefixing `%%` to the label name. So you can invent an instruction which executes a `RET` if the `Z` flag is set by doing this:

```
%macro  retz 0

        jnz     %%skip
        ret
    %%skip:

%endmacro
```

You can call this macro as many times as you want, and every time you call it NASM will make up a different ‘real’ name to substitute for the label `%%skip`. The names NASM invents are of the form `..@2345.skip`, where the number 2345 changes with every macro call. The `..@` prefix prevents macro−local labels from interfering with the local label mechanism, as described in section 3.9. You should avoid defining your own labels in this form (the `..@` prefix, then a number, then another period) in case they interfere with macro−local labels.

### 4.3.3 Greedy Macro Parameters

Occasionally it is useful to define a macro which lumps its entire command line into one parameter definition, possibly after extracting one or two smaller parameters from the front. An example might be a macro to write a text string to a file in MS−DOS, where you might want to be able to write

```
        writefile [filehandle],"hello, world",13,10
```

NASM allows you to define the last parameter of a macro to be *greedy*, meaning that if you invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one along with the separating commas. So if you code:

```
%macro  writefile 2+

        jmp     %%endstr
```

```
 %%str:     db  %2
 %%endstr:
    mov dx,%%str
    mov cx,%%endstr−%%str
    mov bx,%1
    mov ah,0x40
    int 0x21

%endmacro
```

then the example call to `writefile` above will work as expected: the text before the first comma, `[filehandle]`, is used as the first macro parameter and expanded when `%1` is referred to, and all the subsequent text is lumped into `%2` and placed after the `db`.

The greedy nature of the macro is indicated to NASM by the use of the `+` sign after the parameter count on the `%macro` line.

If you define a greedy macro, you are effectively telling NASM how it should expand the macro given *any* number of parameters from the actual number specified up to infinity; in this case, for example, NASM now knows what to do when it sees a call to `writefile` with 2, 3, 4 or more parameters. NASM will take this into account when overloading macros, and will not allow you to define another form of `writefile` taking 4 parameters (for example).

Of course, the above macro could have been implemented as a non−greedy macro, in which case the call to it would have had to look like

```
          writefile [filehandle], @\{"hello, world",13,10@\}
```

NASM provides both mechanisms for putting commas in macro parameters, and you choose which one you prefer for each macro definition.

See section 5.3.1 for a better way to write the above macro.

### 4.3.4 Default Macro Parameters

NASM also allows you to define a multi−line macro with a *range* of allowable parameter counts. If you do this, you can specify defaults for omitted parameters. So, for example:

```
%macro  die 0−1 "Painful program death has occurred."

        writefile 2,%1
        mov     ax,0x4c01
        int     0x21

%endmacro
```

This macro (which makes use of the `writefile` macro defined in section 4.3.3) can be called with an explicit error message, which it will display on the error output stream before exiting, or it can be called with no parameters, in which case it will use the default error message supplied in the macro definition.

In general, you supply a minimum and maximum number of parameters for a macro of this type; the minimum number of parameters are then required in the macro call, and then you provide defaults for the optional ones. So if a macro definition began with the line

```
%macro foobar 1−3 eax,[ebx+2]
```

then it could be called with between one and three parameters, and `%1` would always be taken from the macro call. `%2`, if not specified by the macro call, would default to `eax`, and `%3` if not specified would default to `[ebx+2]`.

You may omit parameter defaults from the macro definition, in which case the parameter default is taken to be blank. This can be useful for macros which can take a variable number of parameters, since the `%0` token (see section 4.3.5) allows you to determine how many parameters were really passed to the macro call.

This defaulting mechanism can be combined with the greedy−parameter mechanism; so the `die` macro above could be made more powerful, and more useful, by changing the first line of the definition to

```
%macro die 0−1+ "Painful program death has occurred.",13,10
```

The maximum parameter count can be infinite, denoted by `*`. In this case, of course, it is impossible to provide a *full* set of default parameters. Examples of this usage are shown in section 4.3.6.

### 4.3.5 `%0`: Macro Parameter Counter

For a macro which can take a variable number of parameters, the parameter reference `%0` will return a numeric constant giving the number of parameters passed to the macro. This can be used as an argument to `%rep` (see section 4.5) in order to iterate through all the parameters of a macro. Examples are given in section 4.3.6.

### 4.3.6 `%rotate`: Rotating Macro Parameters

Unix shell programmers will be familiar with the `shift` shell command, which allows the arguments passed to a shell script (referenced as `$1`, `$2` and so on) to be moved left by one place, so that the argument previously referenced as `$2` becomes available as `$1`, and the argument previously referenced as `$1` is no longer available at all.

NASM provides a similar mechanism, in the form of `%rotate`. As its name suggests, it differs from the Unix `shift` in that no parameters are lost: parameters rotated off the left end of the argument list reappear on the right, and vice versa.

`%rotate` is invoked with a single numeric argument (which may be an expression). The macro parameters are rotated to the left by that many places. If the argument to `%rotate` is negative, the macro parameters are rotated to the right.

So a pair of macros to save and restore a set of registers might work as follows:

```
%macro  multipush 1−*

  %rep  %0
        push    %1
  %rotate 1
  %endrep

%endmacro
```

This macro invokes the `PUSH` instruction on each of its arguments in turn, from left to right. It begins by pushing its first argument, `%1`, then invokes `%rotate` to move all the arguments one place to the left, so that the original second argument is now available as `%1`. Repeating this procedure as many times as there were arguments (achieved by supplying `%0` as the argument to `%rep`) causes each argument in turn to be pushed.

Note also the use of `*` as the maximum parameter count, indicating that there is no upper limit on the number of parameters you may supply to the `multipush` macro.

It would be convenient, when using this macro, to have a `POP` equivalent, which *didn’t* require the arguments to be given in reverse order. Ideally, you would write the `multipush` macro call, then cut−and−paste the line to where the pop needed to be done, and change the name of the called macro to `multipop`, and the macro would take care of popping the registers in the opposite order from the one in which they were pushed.

This can be done by the following definition:

```
%macro  multipop 1-*

  %rep %0
  %rotate -1
        pop     %1
  %endrep

%endmacro
```

This macro begins by rotating its arguments one place to the *right*, so that the original *last* argument appears as `%1`. This is then popped, and the arguments are rotated right again, so the second−to−last argument becomes `%1`. Thus the arguments are iterated through in reverse order.

### 4.3.7 Concatenating Macro Parameters

NASM can concatenate macro parameters on to other text surrounding them. This allows you to declare a family of symbols, for example, in a macro definition. If, for example, you wanted to generate a table of key codes along with offsets into the table, you could code something like

```
%macro keytab_entry 2

    keypos%1    equ     $-keytab
                db      %2

%endmacro

keytab:
          keytab_entry F1,128+1
          keytab_entry F2,128+2
          keytab_entry Return,13
```

which would expand to

```
keytab:
keyposF1        equ     $-keytab
                db     128+1
keyposF2        equ     $-keytab
                db      128+2
keyposReturn    equ     $-keytab
                db      13
```

You can just as easily concatenate text on to the other end of a macro parameter, by writing `%1foo`.

If you need to append a *digit* to a macro parameter, for example defining labels `foo1` and `foo2` when passed the parameter `foo`, you can’t code `%11` because that would be taken as the eleventh macro parameter. Instead, you must code `%{1}1`, which will separate the first `1` (giving the number of the macro parameter) from the second (literal text to be concatenated to the parameter).

This concatenation can also be applied to other preprocessor in−line objects, such as macro−local labels (section 4.3.2) and context−local labels (section 4.7.2). In all cases, ambiguities in syntax can be resolved by enclosing everything after the `%` sign and before the literal text in braces: so `%{%foo}bar` concatenates the text `bar` to the end of the real name of the macro−local label `%%foo`. (This is unnecessary, since the form NASM uses for the real names of macro−local labels means that the two usages `%{%foo}bar` and `%%foobar` would both expand to the same thing anyway; nevertheless, the capability is there.)

### 4.3.8 Condition Codes as Macro Parameters

NASM can give special treatment to a macro parameter which contains a condition code. For a start, you can refer to the macro parameter `%1` by means of the alternative syntax `%+1`, which informs NASM that this macro parameter is supposed to contain a condition code, and will cause the preprocessor to report an error message if the macro is called with a parameter which is *not* a valid condition code.

Far more usefully, though, you can refer to the macro parameter by means of `%−1`, which NASM will expand as the *inverse* condition code. So the `retz` macro defined in section 4.3.2 can be replaced by a general conditional−return macro like this:

```
%macro  retc 1

        j%−1    %%skip
        ret
  %%skip:

%endmacro
```

This macro can now be invoked using calls like `retc ne`, which will cause the conditional−jump instruction in the macro expansion to come out as `JE`, or `retc po` which will make the jump a `JPE`.

The `%+1` macro−parameter reference is quite happy to interpret the arguments `CXZ` and `ECXZ` as valid condition codes; however, `%−1` will report an error if passed either of these, because no inverse condition code exists.

### 4.3.9 Disabling Listing Expansion

When NASM is generating a listing file from your program, it will generally expand multi−line macros by means of writing the macro call and then listing each line of the expansion. This allows you to see which instructions in the macro expansion are generating what code; however, for some macros this clutters the listing up unnecessarily.

NASM therefore provides the `.nolist` qualifier, which you can include in a macro definition to inhibit the expansion of the macro in the listing file. The `.nolist` qualifier comes directly after the number of parameters, like this:

```
%macro foo 1.nolist
```

Or like this:

```
%macro bar 1−5+.nolist a,b,c,d,e,f,g,h
```

## 4.4 Conditional Assembly

Similarly to the C preprocessor, NASM allows sections of a source file to be assembled only if certain conditions are met. The general syntax of this feature looks like this:

```
%if<condition>
    ; some code which only appears if <condition> is met
%elif<condition2>
    ; only appears if <condition> is not met but <condition2> is
%else
    ; this appears if neither <condition> nor <condition2> was met
%endif
```

The `%else` clause is optional, as is the `%elif` clause. You can have more than one `%elif` clause as well.

### 4.4.1 `%ifdef`: Testing Single−Line Macro Existence

Beginning a conditional−assembly block with the line `%ifdef MACRO` will assemble the subsequent code if, and only if, a single−line macro called `MACRO` is defined. If not, then the `%elif` and `%else` blocks (if any) will be processed instead.

For example, when debugging a program, you might want to write code such as

```
          ; perform some function
%ifdef DEBUG
          writefile 2,"Function performed successfully",13,10
%endif
          ; go and do something else
```

Then you could use the command−line option `−dDEBUG` to create a version of the program which produced debugging messages, and remove the option to generate the final release version of the program.

You can test for a macro *not* being defined by using `%ifndef` instead of `%ifdef`. You can also test for macro definitions in `%elif` blocks by using `%elifdef` and `%elifndef`.

### 4.4.2 `ifmacro`: Testing Multi−Line Macro Existence

The `%ifmacro` directive operates in the same way as the `%ifdef` directive, except that it checks for the existence of a multi−line macro.

For example, you may be working with a large project and not have control over the macros in a library. You may want to create a macro with one name if it doesn't already exist, and another name if one with that name does exist.

The `%ifmacro` is considered true if defining a macro with the given name and number of arguments would cause a definitions conflict. For example:

```
%ifmacro MyMacro 1−3

     %error "MyMacro 1−3" causes a conflict with an existing macro.

%else

     %macro MyMacro 1−3

             ; insert code to define the macro

     %endmacro

%endif
```

This will create the macro "MyMacro 1−3" if no macro already exists which would conflict with it, and emits a warning if there would be a definition conflict.

You can test for the macro not existing by using the `%ifnmacro` instead of `%ifmacro`. Additional tests can be performed in `%elif` blocks by using `%elifmacro` and `%elifnmacro`.

### 4.4.3 `%ifctx`: Testing the Context Stack

The conditional−assembly construct `%ifctx ctxname` will cause the subsequent code to be assembled if and only if the top context on the preprocessor's context stack has the name `ctxname`. As with `%ifdef`, the inverse and `%elif` forms `%ifnctx`, `%elifctx` and `%elifnctx` are also supported.

For more details of the context stack, see section 4.7. For a sample use of `%ifctx`, see section 4.7.5.

### 4.4.4 `%if`: Testing Arbitrary Numeric Expressions

The conditional−assembly construct `%if expr` will cause the subsequent code to be assembled if and only if the value of the numeric expression `expr` is non−zero. An example of the use of this feature is in deciding when to break out of a `%rep` preprocessor loop: see section 4.5 for a detailed example.

The expression given to `%if`, and its counterpart `%elif`, is a critical expression (see section 3.8).

`%if` extends the normal NASM expression syntax, by providing a set of relational operators which are not normally available in expressions. The operators `=`, `<`, `>`, `<=`, `>=` and `<>` test equality, less−than, greater−than, less−or−equal, greater−or−equal and not−equal respectively. The C−like forms `==` and `!=` are supported as alternative forms of `=` and `<>`. In addition, low−priority logical operators `&&`, `^^` and `||` are provided, supplying logical AND, logical XOR and logical OR. These work like the C logical operators (although C has no logical XOR), in that they always return either 0 or 1, and treat any non−zero input as 1 (so that `^^`, for example, returns 1 if exactly one of its inputs is zero, and 0 otherwise). The relational operators also return 1 for true and 0 for false.

Like most other `%if` constructs, `%if` has a counterpart `%elif`, and negative forms `%ifn` and `%elifn`.

### 4.4.5 `%ifidn` and `%ifidni`: Testing Exact Text Identity

The construct `%ifidn text1,text2` will cause the subsequent code to be assembled if and only if `text1` and `text2`, after expanding single−line macros, are identical pieces of text. Differences in white space are not counted.

`%ifidni` is similar to `%ifidn`, but is case−insensitive.

For example, the following macro pushes a register or number on the stack, and allows you to treat `IP` as a real register:

```
%macro  pushparam 1

  %ifidni %1,ip
        call    %%label
  %%label:
  %else
        push    %1
  %endif

%endmacro
```

Like most other `%if` constructs, `%ifidn` has a counterpart `%elifidn`, and negative forms `%ifnidn` and `%elifnidn`. Similarly, `%ifidni` has counterparts `%elifidni`, `%ifnidni` and `%elifnidni`.

### 4.4.6 `%ifid`, `%ifnum`, `%ifstr`: Testing Token Types

Some macros will want to perform different tasks depending on whether they are passed a number, a string, or an identifier. For example, a string output macro might want to be able to cope with being passed either a string constant or a pointer to an existing string.

The conditional assembly construct `%ifid`, taking one parameter (which may be blank), assembles the subsequent code if and only if the first token in the parameter exists and is an identifier. `%ifnum` works similarly, but tests for the token being a numeric constant; `%ifstr` tests for it being a string.

For example, the `writefile` macro defined in section 4.3.3 can be extended to take advantage of `%ifstr` in the following fashion:

```
%macro writefile 2−3+

  %ifstr %2
        jmp     %%endstr
    %if %0 = 3
      %%str:    db      %2,%3
    %else
      %%str:    db      %2
    %endif
      %%endstr: mov     dx,%%str
                mov     cx,%%endstr−%%str
  %else
                mov     dx,%2
                mov     cx,%3
  %endif
                mov     bx,%1
                mov     ah,0x40
                int     0x21

%endmacro
```

Then the `writefile` macro can cope with being called in either of the following two ways:

```
        writefile [file], strpointer, length
        writefile [file], "hello", 13, 10
```

In the first, `strpointer` is used as the address of an already−declared string, and `length` is used as its length; in the second, a string is given to the macro, which therefore declares it itself and works out the address and length for itself.

Note the use of `%if` inside the `%ifstr`: this is to detect whether the macro was passed two arguments (so the string would be a single string constant, and `db %2` would be adequate) or more (in which case, all but the first two would be lumped together into `%3`, and `db %2,%3` would be required).

The usual `%elifXXX`, `%ifnXXX` and `%elifnXXX` versions exist for each of `%ifid`, `%ifnum` and `%ifstr`.

### 4.4.7 `%error`: Reporting User−Defined Errors

The preprocessor directive `%error` will cause NASM to report an error if it occurs in assembled code. So if other users are going to try to assemble your source files, you can ensure that they define the right macros by means of code like this:

```
%ifdef SOME_MACRO
    ; do some setup
%elifdef SOME_OTHER_MACRO
    ; do some different setup
%else
```

```
    %error Neither SOME_MACRO nor SOME_OTHER_MACRO was defined.
%endif
```

Then any user who fails to understand the way your code is supposed to be assembled will be quickly warned of their mistake, rather than having to wait until the program crashes on being run and then not knowing what went wrong.

## 4.5 Preprocessor Loops: `%rep`

NASM's `TIMES` prefix, though useful, cannot be used to invoke a multi−line macro multiple times, because it is processed by NASM after macros have already been expanded. Therefore NASM provides another form of loop, this time at the preprocessor level: `%rep`.

The directives `%rep` and `%endrep` (`%rep` takes a numeric argument, which can be an expression; `%endrep` takes no arguments) can be used to enclose a chunk of code, which is then replicated as many times as specified by the preprocessor:

```
%assign i 0
%rep    64
        inc     word [table+2*i]
%assign i i+1
%endrep
```

This will generate a sequence of 64 `INC` instructions, incrementing every word of memory from `[table]` to `[table+126]`.

For more complex termination conditions, or to break out of a repeat loop part way along, you can use the `%exitrep` directive to terminate the loop, like this:

```
fibonacci:
%assign i 0
%assign j 1
%rep 100
%if j > 65535
    %exitrep
%endif
        dw j
%assign k j+i
%assign i j
%assign j k
%endrep

fib_number equ ($−fibonacci)/2
```

This produces a list of all the Fibonacci numbers that will fit in 16 bits. Note that a maximum repeat count must still be given to `%rep`. This is to prevent the possibility of NASM getting into an infinite loop in the preprocessor, which (on multitasking or multi−user systems) would typically cause all the system memory to be gradually used up and other applications to start crashing.

## 4.6 Including Other Files

Using, once again, a very similar syntax to the C preprocessor, NASM's preprocessor lets you include other source files into your code. This is done by the use of the `%include` directive:

```
%include "macros.mac"
```

will include the contents of the file `macros.mac` into the source file containing the `%include` directive.

Include files are searched for in the current directory (the directory you're in when you run NASM, as opposed to the location of the NASM executable or the location of the source file), plus any directories specified on the NASM command line using the `−i` option.

The standard C idiom for preventing a file being included more than once is just as applicable in NASM: if the file `macros.mac` has the form

```
%ifndef MACROS_MAC
    %define MACROS_MAC
    ; now define some macros
%endif
```

then including the file more than once will not cause errors, because the second time the file is included nothing will happen because the macro `MACROS_MAC` will already be defined.

You can force a file to be included even if there is no `%include` directive that explicitly includes it, by using the `−p` option on the NASM command line (see section 2.1.12).

## 4.7 The Context Stack

Having labels that are local to a macro definition is sometimes not quite powerful enough: sometimes you want to be able to share labels between several macro calls. An example might be a `REPEAT` ... `UNTIL` loop, in which the expansion of the `REPEAT` macro would need to be able to refer to a label which the `UNTIL` macro had defined. However, for such a macro you would also want to be able to nest these loops.

NASM provides this level of power by means of a *context stack*. The preprocessor maintains a stack of *contexts*, each of which is characterized by a name. You add a new context to the stack using the `%push` directive, and remove one using `%pop`. You can define labels that are local to a particular context on the stack.

### 4.7.1 `%push` and `%pop`: Creating and Removing Contexts

The `%push` directive is used to create a new context and place it on the top of the context stack. `%push` requires one argument, which is the name of the context. For example:

```
%push    foobar
```

This pushes a new context called `foobar` on the stack. You can have several contexts on the stack with the same name: they can still be distinguished.

The directive `%pop`, requiring no arguments, removes the top context from the context stack and destroys it, along with any labels associated with it.

### 4.7.2 Context−Local Labels

Just as the usage `%%foo` defines a label which is local to the particular macro call in which it is used, the usage `%$foo` is used to define a label which is local to the context on the top of the context stack. So the `REPEAT` and `UNTIL` example given above could be implemented by means of:

```
%macro repeat 0

    %push   repeat
    %$begin:

%endmacro

%macro until 1

        j%−1    %$begin
    %pop

%endmacro
```

and invoked by means of, for example,

```
        mov     cx,string
        repeat
        add     cx,3
        scasb
        until   e
```

which would scan every fourth byte of a string in search of the byte in `AL`.

If you need to define, or access, labels local to the context *below* the top one on the stack, you can use `%$$foo`, or `%$$$foo` for the context below that, and so on.

### 4.7.3 Context−Local Single−Line Macros

NASM also allows you to define single−line macros which are local to a particular context, in just the same way:

```
%define %$localmac 3
```

will define the single−line macro `%$localmac` to be local to the top context on the stack. Of course, after a subsequent `%push`, it can then still be accessed by the name `%$$localmac`.

### 4.7.4 `%repl`: Renaming a Context

If you need to change the name of the top context on the stack (in order, for example, to have it respond differently to `%ifctx`), you can execute a `%pop` followed by a `%push`; but this will have the side effect of destroying all context−local labels and macros associated with the context that was just popped.

NASM provides the directive `%repl`, which *replaces* a context with a different name, without touching the associated macros and labels. So you could replace the destructive code

```
%pop
%push   newname
```

with the non−destructive version `%repl newname`.

### 4.7.5 Example Use of the Context Stack: Block IFs

This example makes use of almost all the context−stack features, including the conditional−assembly construct `%ifctx`, to implement a block IF statement as a set of macros.

```
%macro if 1

    %push if
    j%-1  %$ifnot

%endmacro

%macro else 0

  %ifctx if
        %repl   else
        jmp     %$ifend
        %$ifnot:
```

```
  %else
        %error  "expected ‘if’ before ‘else’"
  %endif

%endmacro

%macro endif 0

  %ifctx if
        %$ifnot:
        %pop
  %elifctx      else
        %$ifend:
        %pop
  %else
        %error  "expected ‘if’ or ‘else’ before ‘endif’"
  %endif

%endmacro
```

This code is more robust than the `REPEAT` and `UNTIL` macros given in section 4.7.2, because it uses conditional assembly to check that the macros are issued in the right order (for example, not calling `endif` before `if`) and issues a `%error` if they're not.

In addition, the `endif` macro has to be able to cope with the two distinct cases of either directly following an `if`, or following an `else`. It achieves this, again, by using conditional assembly to do different things depending on whether the context on top of the stack is `if` or `else`.

The `else` macro has to preserve the context on the stack, in order to have the `%$ifnot` referred to by the `if` macro be the same as the one defined by the `endif` macro, but has to change the context's name so that `endif` will know there was an intervening `else`. It does this by the use of `%repl`.

A sample usage of these macros might look like:

```
        cmp     ax,bx

        if ae
               cmp     bx,cx

               if ae
                       mov     ax,cx
               else
                       mov     ax,bx
               endif

        else
               cmp     ax,cx

               if ae
                       mov     ax,cx
               endif

        endif
```

The block−`IF` macros handle nesting quite happily, by means of pushing another context, describing the inner `if`, on top of the one describing the outer `if`; thus `else` and `endif` always refer to the last unmatched `if` or `else`.

## 4.8 Standard Macros

NASM defines a set of standard macros, which are already defined when it starts to process any source file. If you really need a program to be assembled with no pre−defined macros, you can use the `%clear` directive to empty the preprocessor of everything but context−local preprocessor variables and single−line macros.

Most user−level assembler directives (see chapter 5) are implemented as macros which invoke primitive directives; these are described in chapter 5. The rest of the standard macro set is described here.

### 4.8.1 `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `____NASM_PATCHLEVEL__`: NASM Version

The single−line macros `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `____NASM_PATCHLEVEL__` expand to the major, minor, subminor and patch level parts of the version number of NASM being used. So, under NASM 0.98.32p1 for example, `__NASM_MAJOR__` would be defined to be 0, `__NASM_MINOR__` would be defined as 98, `__NASM_SUBMINOR__` would be defined to 32, and `____NASM_PATCHLEVEL__` would be defined as 1.

### 4.8.2 `__NASM_VERSION_ID__`: NASM Version ID

The single−line macro `__NASM_VERSION_ID__` expands to a dword integer representing the full version number of the version of nasm being used. The value is the equivalent to `__NASM_MAJOR__`, `__NASM_MINOR__`, `__NASM_SUBMINOR__` and `____NASM_PATCHLEVEL__` concatenated to produce a single doubleword. Hence, for 0.98.32p1, the returned number would be equivalent to:

```
        dd      0x00622001
```

or

```
        db      1,32,98,0
```

Note that the above lines are generate exactly the same code, the second line is used just to give an indication of the order that the separate values will be present in memory.

### 4.8.3 `__NASM_VER__`: NASM Version string

The single−line macro `__NASM_VER__` expands to a string which defines the version number of nasm being used. So, under NASM 0.98.32 for example,

```
        db      __NASM_VER__
```

would expand to

```
        db      "0.98.32"
```

### 4.8.4 `__FILE__` and `__LINE__`: File Name and Line Number

Like the C preprocessor, NASM allows the user to find out the file name and line number containing the current instruction. The macro `__FILE__` expands to a string constant giving the name of the current input file (which may change through the course of assembly if `%include` directives are used), and `__LINE__` expands to a numeric constant giving the current line number in the input file.

These macros could be used, for example, to communicate debugging information to a macro, since invoking `__LINE__` inside a macro definition (either single−line or multi−line) will return the line number of the macro *call*, rather than *definition*. So to determine where in a piece of code a crash is occurring, for example, one could write a routine `stillhere`, which is passed a line number in `EAX` and outputs something like ‘line 155: still here’. You could then write a macro

```
%macro  notdeadyet 0

        push    eax
        mov     eax,__LINE__
        call    stillhere
        pop     eax

%endmacro
```

and then pepper your code with calls to `notdeadyet` until you find the crash point.

### 4.8.5 `__BITS__`: Current BITS Mode

The `__BITS__` standard macro is updated every time that the BITS mode is set using the `BITS XX` or `[BITS XX]` directive, where XX is a valid mode number of 16, 32 or 64. `__BITS__` receives the specified mode number and makes it globally available. This can be very useful for those who utilize mode−dependent macros.

### 4.8.6 `STRUC` and `ENDSTRUC`: Declaring Structure Data Types

The core of NASM contains no intrinsic means of defining data structures; instead, the preprocessor is sufficiently powerful that data structures can be implemented as a set of macros. The macros `STRUC` and `ENDSTRUC` are used to define a structure data type.

`STRUC` takes one parameter, which is the name of the data type. This name is defined as a symbol with the value zero, and also has the suffix `_size` appended to it and is then defined as an `EQU` giving the size of the structure. Once `STRUC` has been issued, you are defining the structure, and should define fields using the `RESB` family of pseudo−instructions, and then invoke `ENDSTRUC` to finish the definition.

For example, to define a structure called `mytype` containing a longword, a word, a byte and a string of bytes, you might code

```
struc   mytype

  mt_long:      resd    1
  mt_word:      resw    1
  mt_byte:      resb    1
  mt_str:       resb    32

endstruc
```

The above code defines six symbols: `mt_long` as 0 (the offset from the beginning of a `mytype` structure to the longword field), `mt_word` as 4, `mt_byte` as 6, `mt_str` as 7, `mytype_size` as 39, and `mytype` itself as zero.

The reason why the structure type name is defined at zero is a side effect of allowing structures to work with the local label mechanism: if your structure members tend to have the same names in more than one structure, you can define the above structure like this:

```
struc mytype

  .long:        resd    1
  .word:        resw    1
  .byte:        resb    1
  .str:         resb    32

endstruc
```

This defines the offsets to the structure fields as `mytype.long`, `mytype.word`, `mytype.byte` and `mytype.str`.

NASM, since it has no *intrinsic* structure support, does not support any form of period notation to refer to the elements of a structure once you have one (except the above local−label notation), so code such as `mov ax,[mystruc.mt_word]` is not valid. `mt_word` is a constant just like any other constant, so the correct syntax is `mov ax,[mystruc+mt_word]` or `mov ax,[mystruc+mytype.word]`.

### 4.8.7 `ISTRUC`, `AT` and `IEND`: Declaring Instances of Structures

Having defined a structure type, the next thing you typically want to do is to declare instances of that structure in your data segment. NASM provides an easy way to do this in the `ISTRUC` mechanism. To declare a structure of type `mytype` in a program, you code something like this:

```
mystruc:
    istruc mytype

        at mt_long, dd      123456
        at mt_word, dw      1024
        at mt_byte, db      'x'
        at mt_str,  db      'hello, world', 13, 10, 0

    iend
```

The function of the `AT` macro is to make use of the `TIMES` prefix to advance the assembly position to the correct point for the specified structure field, and then to declare the specified data. Therefore the structure fields must be declared in the same order as they were specified in the structure definition.

If the data to go in a structure field requires more than one source line to specify, the remaining source lines can easily come after the `AT` line. For example:

```
        at mt_str,  db      123,134,145,156,167,178,189
                    db      190,100,0
```

Depending on personal taste, you can also omit the code part of the `AT` line completely, and start the structure field on the next line:

```
        at mt_str
                db      'hello, world'
                db      13,10,0
```

### 4.8.8 `ALIGN` and `ALIGNB`: Data Alignment

The `ALIGN` and `ALIGNB` macros provides a convenient way to align code or data on a word, longword, paragraph or other boundary. (Some assemblers call this directive `EVEN`.) The syntax of the `ALIGN` and `ALIGNB` macros is

```
        align   4               ; align on 4−byte boundary
        align   16              ; align on 16−byte boundary
        align   8,db 0          ; pad with 0s rather than NOPs
        align   4,resb 1        ; align to 4 in the BSS
        alignb  4               ; equivalent to previous line
```

Both macros require their first argument to be a power of two; they both compute the number of additional bytes required to bring the length of the current section up to a multiple of that power of two, and then apply the `TIMES` prefix to their second argument to perform the alignment.

If the second argument is not specified, the default for `ALIGN` is `NOP`, and the default for `ALIGNB` is `RESB 1`. So if the second argument is specified, the two macros are equivalent. Normally, you can just use `ALIGN` in code and data sections and `ALIGNB` in BSS sections, and never need the second argument except for special purposes.

`ALIGN` and `ALIGNB`, being simple macros, perform no error checking: they cannot warn you if their first argument fails to be a power of two, or if their second argument generates more than one byte of code. In each of these cases they will silently do the wrong thing.

`ALIGNB` (or `ALIGN` with a second argument of `RESB 1`) can be used within structure definitions:

```
struc mytype2

  mt_byte:
        resb 1
        alignb 2
  mt_word:
        resw 1
        alignb 4
  mt_long:
        resd 1
  mt_str:
        resb 32

endstruc
```

This will ensure that the structure members are sensibly aligned relative to the base of the structure.

A final caveat: `ALIGN` and `ALIGNB` work relative to the beginning of the *section*, not the beginning of the address space in the final executable. Aligning to a 16−byte boundary when the section you're in is only guaranteed to be aligned to a 4−byte boundary, for example, is a waste of effort. Again, NASM does not check that the section's alignment characteristics are sensible for the use of `ALIGN` or `ALIGNB`.

## 4.9 TASM Compatible Preprocessor Directives

The following preprocessor directives may only be used when TASM compatibility is turned on using the `−t` command line switch (This switch is described in section 2.1.18.)

- `%arg` (see section 4.9.1)
- `%stacksize` (see section 4.9.2)
- `%local` (see section 4.9.3)

### 4.9.1 `%arg` Directive

The `%arg` directive is used to simplify the handling of parameters passed on the stack. Stack based parameter passing is used by many high level languages, including C, C++ and Pascal.

While NASM comes with macros which attempt to duplicate this functionality (see section 7.4.5), the syntax is not particularly convenient to use and is not TASM compatible. Here is an example which shows the use of `%arg` without any external macros:

```
some_function:

    %push     mycontext        ; save the current context
    %stacksize large           ; tell NASM to use bp
    %arg      i:word, j_ptr:word

        mov     ax,[i]
        mov     bx,[j_ptr]
        add     ax,[bx]
        ret

    %pop                       ; restore original context
```

This is similar to the procedure defined in section 7.4.5 and adds the value in i to the value pointed to by j_ptr and returns the sum in the ax register. See section 4.7.1 for an explanation of `push` and `pop` and the use of context stacks.

### 4.9.2 `%stacksize` Directive

The `%stacksize` directive is used in conjunction with the `%arg` (see section 4.9.1) and the `%local` (see section 4.9.3) directives. It tells NASM the default size to use for subsequent `%arg` and `%local` directives. The `%stacksize` directive takes one required argument which is one of `flat`, `large` or `small`.

```
%stacksize flat
```

This form causes NASM to use stack−based parameter addressing relative to `ebp` and it assumes that a near form of call was used to get to this label (i.e. that `eip` is on the stack).

```
%stacksize large
```

This form uses `bp` to do stack−based parameter addressing and assumes that a far form of call was used to get to this address (i.e. that `ip` and `cs` are on the stack).

```
%stacksize small
```

This form also uses `bp` to address stack parameters, but it is different from `large` because it also assumes that the old value of bp is pushed onto the stack (i.e. it expects an `ENTER` instruction). In other words, it expects that `bp`, `ip` and `cs` are on the top of the stack, underneath any local space which may have been allocated by `ENTER`. This form is probably most useful when used in combination with the `%local` directive (see section 4.9.3).

### 4.9.3 `%local` Directive

The `%local` directive is used to simplify the use of local temporary stack variables allocated in a stack frame. Automatic local variables in C are an example of this kind of variable. The `%local` directive is most useful when used with the `%stacksize` (see section 4.9.2 and is also compatible with the `%arg` directive (see section 4.9.1). It allows simplified reference to variables on the stack which have been allocated typically by using the `ENTER` instruction. An example of its use is the following:

```
silly_swap:

    %push mycontext             ; save the current context
    %stacksize small            ; tell NASM to use bp
    %assign %$localsize 0       ; see text for explanation
    %local old_ax:word, old_dx:word

        enter   %$localsize,0   ; see text for explanation
        mov     [old_ax],ax     ; swap ax & bx
        mov     [old_dx],dx     ; and swap dx & cx
        mov     ax,bx
        mov     dx,cx
        mov     bx,[old_ax]
        mov     cx,[old_dx]
        leave                   ; restore old bp
        ret                     ;

    %pop                        ; restore original context
```

The `%$localsize` variable is used internally by the `%local` directive and *must* be defined within the current context before the `%local` directive may be used. Failure to do so will result in one expression syntax error for each `%local` variable declared. It then may be used in the construction of an appropriately sized ENTER instruction as shown in the example.

## 4.10 Other Preprocessor Directives

NASM also has preprocessor directives which allow access to information from external sources. Currently they include:

The following preprocessor directive is supported to allow NASM to correctly handle output of the cpp C language preprocessor.

- `%line` enables NAsM to correctly handle the output of the cpp C language preprocessor (see section 4.10.1).
- `%!` enables NASM to read in the value of an environment variable, which can then be used in your program (see section 4.10.2).

### 4.10.1 `%line` Directive

The `%line` directive is used to notify NASM that the input line corresponds to a specific line number in another file. Typically this other file would be an original source file, with the current NASM input being the output of a pre−processor. The `%line` directive allows NASM to output messages which indicate the line number of the original source file, instead of the file that is being read by NASM.

This preprocessor directive is not generally of use to programmers, by may be of interest to preprocessor authors. The usage of the `%line` preprocessor directive is as follows:

```
%line nnn[+mmm] [filename]
```

In this directive, `nnn` indentifies the line of the original source file which this line corresponds to. `mmm` is an optional parameter which specifies a line increment value; each line of the input file read in is considered to correspond to `mmm` lines of the original source file. Finally, `filename` is an optional parameter which specifies the file name of the original source file.

After reading a `%line` preprocessor directive, NASM will report all file name and line numbers relative to the values specified therein.

### 4.10.2 `%!<env>`: Read an environment variable.

The `%!<env>` directive makes it possible to read the value of an environment variable at assembly time. This could, for example, be used to store the contents of an environment variable into a string, which could be used at some other point in your code.

For example, suppose that you have an environment variable `FOO`, and you want the contents of `FOO` to be embedded in your program. You could do that as follows:

```
%define FOO    %!FOO
%define quote   ’

tmpstr  db      quote FOO quote
```

At the time of writing, this will generate an "unterminated string" warning at the time of defining "quote", and it will add a space before and after the string that is read in. I was unable to find a simple workaround (although a workaround can be created using a multi−line macro), so I believe that you will need to either learn how to create more complex macros, or allow for the extra spaces if you make use of this feature in that way.

# Chapter 5: Assembler Directives

NASM, though it attempts to avoid the bureaucracy of assemblers like MASM and TASM, is nevertheless forced to support a *few* directives. These are described in this chapter.

NASM’s directives come in two types: *user−level* directives and *primitive* directives. Typically, each directive has a user−level form and a primitive form. In almost all cases, we recommend that users use the user−level forms of the directives, which are implemented as macros which call the primitive forms.

Primitive directives are enclosed in square brackets; user−level directives are not.

In addition to the universal directives described in this chapter, each object file format can optionally supply extra directives in order to control particular features of that file format. These *format−specific* directives are documented along with the formats that implement them, in chapter 6.

## 5.1 `BITS`: Specifying Target Processor Mode

The `BITS` directive specifies whether NASM should generate code designed to run on a processor operating in 16−bit mode, 32−bit mode or 64−bit mode. The syntax is `BITS XX`, where XX is 16, 32 or 64.

In most cases, you should not need to use `BITS` explicitly. The `aout`, `coff`, `elf`, `macho`, `win32` and `win64` object formats, which are designed for use in 32−bit or 64−bit operating systems, all cause NASM to select 32−bit or 64−bit mode, respectively, by default. The `obj` object format allows you to specify each segment you define as either `USE16` or `USE32`, and NASM will set its operating mode accordingly, so the use of the `BITS` directive is once again unnecessary.

The most likely reason for using the `BITS` directive is to write 32−bit or 64−bit code in a flat binary file; this is because the `bin` output format defaults to 16−bit mode in anticipation of it being used most frequently to write DOS `.COM` programs, DOS `.SYS` device drivers and boot loader software.

You do *not* need to specify `BITS 32` merely in order to use 32−bit instructions in a 16−bit DOS program; if you do, the assembler will generate incorrect code because it will be writing code targeted at a 32−bit platform, to be run on a 16−bit one.

When NASM is in `BITS 16` mode, instructions which use 32−bit data are prefixed with an 0x66 byte, and those referring to 32−bit addresses have an 0x67 prefix. In `BITS 32` mode, the reverse is true: 32−bit instructions require no prefixes, whereas instructions using 16−bit data need an 0x66 and those working on 16−bit addresses need an 0x67.

When NASM is in `BITS 64` mode, most instructions operate the same as they do for `BITS 32` mode. However, there are 8 more general and SSE registers, and 16−bit addressing is no longer supported.

The default address size is 64 bits; 32−bit addressing can be selected with the 0x67 prefix. The default operand size is still 32 bits, however, and the 0x66 prefix selects 16−bit operand size. The `REX` prefix is used both to select 64−bit operand size, and to access the new registers. NASM automatically inserts REX prefixes when necessary.

When the `REX` prefix is used, the processor does not know how to address the AH, BH, CH or DH (high 8−bit legacy) registers. Instead, it is possible to access the the low 8−bits of the SP, BP SI and DI registers as SPL, BPL, SIL and DIL, respectively; but only when the REX prefix is used.

The `BITS` directive has an exactly equivalent primitive form, `[BITS 16]`, `[BITS 32]` and `[BITS 64]`. The user−level form is a macro which has no function other than to call the primitive form.

Note that the space is neccessary, e.g. `BITS32` will *not* work!

### 5.1.1 `USE16` & `USE32`: Aliases for BITS

The '`USE16`' and '`USE32`' directives can be used in place of '`BITS 16`' and '`BITS 32`', for compatibility with other assemblers.

## 5.2 `DEFAULT`: Change the assembler defaults

The `DEFAULT` directive changes the assembler defaults. Normally, NASM defaults to a mode where the programmer is expected to explicitly specify most features directly. However, this is occationally obnoxious, as the explicit form is pretty much the only one one wishes to use.

Currently, the only `DEFAULT` that is settable is whether or not registerless instructions in 64−bit mode are `RIP`–relative or not. By default, they are absolute unless overridden with the `REL` specifier (see section 3.3). However, if `DEFAULT REL` is specified, `REL` is default, unless overridden with the `ABS` specifier, *except when used with an FS or GS segment override*.

The special handling of `FS` and `GS` overrides are due to the fact that these registers are generally used as thread pointers or other special functions in 64−bit mode, and generating `RIP`–relative addresses would be extremely confusing.

`DEFAULT REL` is disabled with `DEFAULT ABS`.

## 5.3 `SECTION` or `SEGMENT`: Changing and Defining Sections

The `SECTION` directive (`SEGMENT` is an exactly equivalent synonym) changes which section of the output file the code you write will be assembled into. In some object file formats, the number and names of sections are fixed; in others, the user may make up as many as they wish. Hence `SECTION` may sometimes give an error message, or may define a new section, if you try to switch to a section that does not (yet) exist.

The Unix object formats, and the `bin` object format (but see section 6.1.3, all support the standardized section names `.text`, `.data` and `.bss` for the code, data and uninitialized−data sections. The `obj` format, by contrast, does not recognize these section names as being special, and indeed will strip off the leading period of any section name that has one.

### 5.3.1 The `__SECT__` Macro

The `SECTION` directive is unusual in that its user−level form functions differently from its primitive form. The primitive form, `[SECTION xyz]`, simply switches the current target section to the one given. The user−level form, `SECTION xyz`, however, first defines the single−line macro `__SECT__` to be the primitive `[SECTION]` directive which it is about to issue, and then issues it. So the user−level directive

```
        SECTION .text
```

expands to the two lines

```
%define __SECT__        [SECTION .text]
        [SECTION .text]
```

Users may find it useful to make use of this in their own macros. For example, the `writefile` macro defined in section 4.3.3 can be usefully rewritten in the following more sophisticated form:

```
%macro  writefile 2+
        [section .data]

  %%str:        db      %2
  %%endstr:

        __SECT__

        mov     dx,%%str
        mov     cx,%%endstr-%%str
        mov     bx,%1
        mov     ah,0x40
        int     0x21

%endmacro
```

This form of the macro, once passed a string to output, first switches temporarily to the data section of the file, using the primitive form of the `SECTION` directive so as not to modify `__SECT__`. It then declares its string in the data section, and then invokes `__SECT__` to switch back to *whichever* section the user was previously working in. It thus avoids the need, in the previous version of the macro, to include a `JMP` instruction to jump over the data, and also does not fail if, in a complicated `OBJ` format module, the user could potentially be assembling the code in any of several separate code sections.

## 5.4 `ABSOLUTE`: Defining Absolute Labels

The `ABSOLUTE` directive can be thought of as an alternative form of `SECTION`: it causes the subsequent code to be directed at no physical section, but at the hypothetical section starting at the given absolute address. The only instructions you can use in this mode are the `RESB` family.

`ABSOLUTE` is used as follows:

```
absolute 0x1A

    kbuf_chr    resw    1
    kbuf_free   resw    1
    kbuf        resw    16
```

This example describes a section of the PC BIOS data area, at segment address 0x40: the above code defines `kbuf_chr` to be 0x1A, `kbuf_free` to be 0x1C, and `kbuf` to be 0x1E.

The user−level form of `ABSOLUTE`, like that of `SECTION`, redefines the `__SECT__` macro when it is invoked.

`STRUC` and `ENDSTRUC` are defined as macros which use `ABSOLUTE` (and also `__SECT__`).

`ABSOLUTE` doesn’t have to take an absolute constant as an argument: it can take an expression (actually, a critical expression: see section 3.8) and it can be a value in a segment. For example, a TSR can re−use its setup code as run−time BSS like this:

```
        org     100h               ; it’s a .COM program

        jmp     setup              ; setup code comes last

        ; the resident part of the TSR goes here
setup:
        ; now write the code that installs the TSR here

absolute setup
```

```
runtimevar1     resw    1
runtimevar2     resd    20

tsr_end:
```

This defines some variables ‘on top of’ the setup code, so that after the setup has finished running, the space it took up can be re−used as data storage for the running TSR. The symbol ‘tsr_end’ can be used to calculate the total size of the part of the TSR that needs to be made resident.

## 5.5 `EXTERN`: Importing Symbols from Other Modules

`EXTERN` is similar to the MASM directive `EXTRN` and the C keyword `extern`: it is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one. Not every object−file format can support external variables: the `bin` format cannot.

The `EXTERN` directive takes as many arguments as you like. Each argument is the name of a symbol:

```
extern  _printf
extern  _sscanf,_fscanf
```

Some object−file formats provide extra features to the `EXTERN` directive. In all cases, the extra features are used by suffixing a colon to the symbol name followed by object−format specific text. For example, the `obj` format allows you to declare that the default segment base of an external should be the group `dgroup` by means of the directive

```
extern  _variable:wrt dgroup
```

The primitive form of `EXTERN` differs from the user−level form only in that it can take only one argument at a time: the support for multiple arguments is implemented at the preprocessor level.

You can declare the same variable as `EXTERN` more than once: NASM will quietly ignore the second and later redeclarations. You can’t declare a variable as `EXTERN` as well as something else, though.

## 5.6 `GLOBAL`: Exporting Symbols to Other Modules

`GLOBAL` is the other end of `EXTERN`: if one module declares a symbol as `EXTERN` and refers to it, then in order to prevent linker errors, some other module must actually *define* the symbol and declare it as `GLOBAL`. Some assemblers use the name `PUBLIC` for this purpose.

The `GLOBAL` directive applying to a symbol must appear *before* the definition of the symbol.

`GLOBAL` uses the same syntax as `EXTERN`, except that it must refer to symbols which *are* defined in the same module as the `GLOBAL` directive. For example:

```
global _main
_main:
        ; some code
```

`GLOBAL`, like `EXTERN`, allows object formats to define private extensions by means of a colon. The `elf` object format, for example, lets you specify whether global data items are functions or data:

```
global  hashlookup:function, hashtable:data
```

Like `EXTERN`, the primitive form of `GLOBAL` differs from the user−level form only in that it can take only one argument at a time.

## 5.7 `COMMON`: Defining Common Data Areas

The `COMMON` directive is used to declare *common variables*. A common variable is much like a global variable declared in the uninitialized data section, so that

```
common  intvar  4
```

is similar in function to

```
global  intvar
section .bss

intvar  resd    1
```

The difference is that if more than one module defines the same common variable, then at link time those variables will be *merged*, and references to `intvar` in all modules will point at the same piece of memory.

Like `GLOBAL` and `EXTERN`, `COMMON` supports object−format specific extensions. For example, the `obj` format allows common variables to be NEAR or FAR, and the `elf` format allows you to specify the alignment requirements of a common variable:

```
common  commvar  4:near  ; works in OBJ
common  intarray 100:4   ; works in ELF: 4 byte aligned
```

Once again, like `EXTERN` and `GLOBAL`, the primitive form of `COMMON` differs from the user−level form only in that it can take only one argument at a time.

## 5.8 `CPU`: Defining CPU Dependencies

The `CPU` directive restricts assembly to those instructions which are available on the specified CPU.

Options are:

- `CPU 8086` Assemble only 8086 instruction set
- `CPU 186` Assemble instructions up to the 80186 instruction set
- `CPU 286` Assemble instructions up to the 286 instruction set
- `CPU 386` Assemble instructions up to the 386 instruction set
- `CPU 486` 486 instruction set
- `CPU 586` Pentium instruction set
- `CPU PENTIUM` Same as 586
- `CPU 686` P6 instruction set
- `CPU PPRO` Same as 686
- `CPU P2` Same as 686
- `CPU P3` Pentium III (Katmai) instruction sets
- `CPU KATMAI` Same as P3
- `CPU P4` Pentium 4 (Willamette) instruction set
- `CPU WILLAMETTE` Same as P4
- `CPU PRESCOTT` Prescott instruction set
- `CPU X64` x86−64 (x64/AMD64/EM64T) instruction set

- `CPU IA64` IA64 CPU (in x86 mode) instruction set

All options are case insensitive. All instructions will be selected only if they apply to the selected CPU or lower. By default, all instructions are available.

## 5.9 `FLOAT`: Handling of floating−point constants

By default, floating−point constants are rounded to nearest, and IEEE denormals are supported. The following options can be set to alter this behaviour:

- `FLOAT DAZ` Flush denormals to zero
- `FLOAT NODAZ` Do not flush denormals to zero (default)
- `FLOAT NEAR` Round to nearest (default)
- `FLOAT UP` Round up (toward +Infinity)
- `FLOAT DOWN` Round down (toward –Infinity)
- `FLOAT ZERO` Round toward zero
- `FLOAT DEFAULT` Restore default settings

The standard macros `__FLOAT_DAZ__`, `__FLOAT_ROUND__`, and `__FLOAT__` contain the current state, as long as the programmer has avoided the use of the brackeded primitive form, (`[FLOAT]`).

`__FLOAT__` contains the full set of floating−point settings; this value can be saved away and invoked later to restore the setting.

# Chapter 6: Output Formats

NASM is a portable assembler, designed to be able to compile on any ANSI C−supporting platform and produce output to run on a variety of Intel x86 operating systems. For this reason, it has a large number of available output formats, selected using the `−f` option on the NASM command line. Each of these formats, along with its extensions to the base NASM syntax, is detailed in this chapter.

As stated in section 2.1.1, NASM chooses a default name for your output file based on the input file name and the chosen output format. This will be generated by removing the extension (`.asm`, `.s`, or whatever you like to use) from the input file name, and substituting an extension defined by the output format. The extensions are given with each format below.

## 6.1 `bin`: Flat−Form Binary Output

The `bin` format does not produce object files: it generates nothing in the output file except the code you wrote. Such ‘pure binary’ files are used by MS−DOS: `.COM` executables and `.SYS` device drivers are pure binary files. Pure binary output is also useful for operating system and boot loader development.

The `bin` format supports multiple section names. For details of how nasm handles sections in the `bin` format, see section 6.1.3.

Using the `bin` format puts NASM by default into 16−bit mode (see section 5.1). In order to use `bin` to write 32−bit or 64−bit code, such as an OS kernel, you need to explicitly issue the `BITS 32` or `BITS 64` directive.

`bin` has no default output file name extension: instead, it leaves your file name as it is once the original extension has been removed. Thus, the default is for NASM to assemble `binprog.asm` into a binary file called `binprog`.

### 6.1.1 `ORG`: Binary File Program Origin

The `bin` format provides an additional directive to the list given in chapter 5: `ORG`. The function of the `ORG` directive is to specify the origin address which NASM will assume the program begins at when it is loaded into memory.

For example, the following code will generate the longword `0x00000104`:

```
        org     0x100
        dd      label
label:
```

Unlike the `ORG` directive provided by MASM−compatible assemblers, which allows you to jump around in the object file and overwrite code you have already generated, NASM’s `ORG` does exactly what the directive says: *origin*. Its sole function is to specify one offset which is added to all internal address references within the section; it does not permit any of the trickery that MASM’s version does. See section 11.1.3 for further comments.

### 6.1.2 `bin` Extensions to the `SECTION` Directive

The `bin` output format extends the `SECTION` (or `SEGMENT`) directive to allow you to specify the alignment requirements of segments. This is done by appending the `ALIGN` qualifier to the end of the section−definition line. For example,

```
section .data   align=16
```

switches to the section `.data` and also specifies that it must be aligned on a 16−byte boundary.

The parameter to `ALIGN` specifies how many low bits of the section start address must be forced to zero. The alignment value given may be any power of two.

### 6.1.3 `Multisection`support for the BIN format.

The `bin` format allows the use of multiple sections, of arbitrary names, besides the "known" `.text`, `.data`, and `.bss` names.

- Sections may be designated `progbits` or `nobits`. Default is `progbits` (except `.bss`, which defaults to `nobits`, of course).
- Sections can be aligned at a specified boundary following the previous section with `align=`, or at an arbitrary byte−granular position with `start=`.
- Sections can be given a virtual start address, which will be used for the calculation of all memory references within that section with `vstart=`.
- Sections can be ordered using `follows=<section>` or `vfollows=<section>` as an alternative to specifying an explicit start address.
- Arguments to `org`, `start`, `vstart`, and `align=` are critical expressions. See section 3.8. E.g. `align=(1 << ALIGN_SHIFT)` – `ALIGN_SHIFT` must be defined before it is used here.
- Any code which comes before an explicit `SECTION` directive is directed by default into the `.text` section.
- If an `ORG` statement is not given, `ORG 0` is used by default.
- The `.bss` section will be placed after the last `progbits` section, unless `start=`, `vstart=`, `follows=`, or `vfollows=` has been specified.
- All sections are aligned on dword boundaries, unless a different alignment has been specified.
- Sections may not overlap.
- Nasm creates the `section.<secname>.start` for each section, which may be used in your code.

### 6.1.4 Map files

Map files can be generated in `-f bin` format by means of the `[map]` option. Map types of `all` (default), `brief`, `sections`, `segments`, or `symbols` may be specified. Output may be directed to `stdout` (default), `stderr`, or a specified file. E.g. `[map symbols myfile.map]`. No "user form" exists, the square brackets must be used.

## 6.2 `obj`: Microsoft OMF Object Files

The `obj` file format (NASM calls it `obj` rather than `omf` for historical reasons) is the one produced by MASM and TASM, which is typically fed to 16−bit DOS linkers to produce `.EXE` files. It is also the format used by OS/2.

`obj` provides a default output file−name extension of `.obj`.

`obj` is not exclusively a 16−bit format, though: NASM has full support for the 32−bit extensions to the format. In particular, 32−bit `obj` format files are used by Borland's Win32 compilers, instead of using Microsoft's newer `win32` object file format.

The `obj` format does not define any special segment names: you can call your segments anything you like. Typical names for segments in `obj` format files are `CODE`, `DATA` and `BSS`.

If your source file contains code before specifying an explicit `SEGMENT` directive, then NASM will invent its own segment called `__NASMDEFSEG` for you.

When you define a segment in an `obj` file, NASM defines the segment name as a symbol as well, so that you can access the segment address of the segment. So, for example:

```
segment data

dvar:   dw      1234

segment code

function:
        mov     ax,data         ; get segment address of data
        mov     ds,ax           ; and move it into DS
        inc     word [dvar]     ; now this reference will work
        ret
```

The `obj` format also enables the use of the `SEG` and `WRT` operators, so that you can write code which does things like

```
extern  foo

      mov   ax,seg foo            ; get preferred segment of foo
      mov   ds,ax
      mov   ax,data               ; a different segment
      mov   es,ax
      mov   ax,[ds:foo]           ; this accesses ‘foo’
      mov   [es:foo wrt data],bx  ; so does this
```

### 6.2.1 `obj` Extensions to the `SEGMENT` Directive

The `obj` output format extends the `SEGMENT` (or `SECTION`) directive to allow you to specify various properties of the segment you are defining. This is done by appending extra qualifiers to the end of the segment−definition line. For example,

```
segment code private align=16
```

defines the segment `code`, but also declares it to be a private segment, and requires that the portion of it described in this code module must be aligned on a 16−byte boundary.

The available qualifiers are:

- `PRIVATE`, `PUBLIC`, `COMMON` and `STACK` specify the combination characteristics of the segment. `PRIVATE` segments do not get combined with any others by the linker; `PUBLIC` and `STACK` segments get concatenated together at link time; and `COMMON` segments all get overlaid on top of each other rather than stuck end−to−end.
- `ALIGN` is used, as shown above, to specify how many low bits of the segment start address must be forced to zero. The alignment value given may be any power of two from 1 to 4096; in reality, the only values supported are 1, 2, 4, 16, 256 and 4096, so if 8 is specified it will be rounded up to 16, and 32, 64 and 128 will all be rounded up to 256, and so on. Note that alignment to 4096−byte boundaries is a PharLap extension to the format and may not be supported by all linkers.
- `CLASS` can be used to specify the segment class; this feature indicates to the linker that segments of the same class should be placed near each other in the output file. The class name can be any word, e.g. `CLASS=CODE`.

- `OVERLAY`, like `CLASS`, is specified with an arbitrary word as an argument, and provides overlay information to an overlay−capable linker.
- Segments can be declared as `USE16` or `USE32`, which has the effect of recording the choice in the object file and also ensuring that NASM's default assembly mode when assembling in that segment is 16−bit or 32−bit respectively.
- When writing OS/2 object files, you should declare 32−bit segments as `FLAT`, which causes the default segment base for anything in the segment to be the special group `FLAT`, and also defines the group if it is not already defined.
- The `obj` file format also allows segments to be declared as having a pre−defined absolute segment address, although no linkers are currently known to make sensible use of this feature; nevertheless, NASM allows you to declare a segment such as `SEGMENT SCREEN ABSOLUTE=0xB800` if you need to. The `ABSOLUTE` and `ALIGN` keywords are mutually exclusive.

NASM's default segment attributes are `PUBLIC`, `ALIGN=1`, no class, no overlay, and `USE16`.

### 6.2.2 `GROUP`: Defining Groups of Segments

The `obj` format also allows segments to be grouped, so that a single segment register can be used to refer to all the segments in a group. NASM therefore supplies the `GROUP` directive, whereby you can code

```
segment data

        ; some data

segment bss

        ; some uninitialized data

group dgroup data bss
```

which will define a group called `dgroup` to contain the segments `data` and `bss`. Like `SEGMENT`, `GROUP` causes the group name to be defined as a symbol, so that you can refer to a variable `var` in the `data` segment as `var wrt data` or as `var wrt dgroup`, depending on which segment value is currently in your segment register.

If you just refer to `var`, however, and `var` is declared in a segment which is part of a group, then NASM will default to giving you the offset of `var` from the beginning of the *group*, not the *segment*. Therefore `SEG var`, also, will return the group base rather than the segment base.

NASM will allow a segment to be part of more than one group, but will generate a warning if you do this. Variables declared in a segment which is part of more than one group will default to being relative to the first group that was defined to contain the segment.

A group does not have to contain any segments; you can still make `WRT` references to a group which does not contain the variable you are referring to. OS/2, for example, defines the special group `FLAT` with no segments in it.

### 6.2.3 `UPPERCASE`: Disabling Case Sensitivity in Output

Although NASM itself is case sensitive, some OMF linkers are not; therefore it can be useful for NASM to output single−case object files. The `UPPERCASE` format−specific directive causes all segment, group and symbol names that are written to the object file to be forced to upper case just before being written. Within a source file, NASM is still case−sensitive; but the object file can be written entirely in upper case if desired.

`UPPERCASE` is used alone on a line; it requires no parameters.

### 6.2.4 `IMPORT`: Importing DLL Symbols

The `IMPORT` format−specific directive defines a symbol to be imported from a DLL, for use if you are writing a DLL's import library in NASM. You still need to declare the symbol as `EXTERN` as well as using the `IMPORT` directive.

The `IMPORT` directive takes two required parameters, separated by white space, which are (respectively) the name of the symbol you wish to import and the name of the library you wish to import it from. For example:

```
import  WSAStartup wsock32.dll
```

A third optional parameter gives the name by which the symbol is known in the library you are importing it from, in case this is not the same as the name you wish the symbol to be known by to your code once you have imported it. For example:

```
import  asyncsel wsock32.dll WSAAsyncSelect
```

### 6.2.5 `EXPORT`: Exporting DLL Symbols

The `EXPORT` format−specific directive defines a global symbol to be exported as a DLL symbol, for use if you are writing a DLL in NASM. You still need to declare the symbol as `GLOBAL` as well as using the `EXPORT` directive.

`EXPORT` takes one required parameter, which is the name of the symbol you wish to export, as it was defined in your source file. An optional second parameter (separated by white space from the first) gives the *external* name of the symbol: the name by which you wish the symbol to be known to programs using the DLL. If this name is the same as the internal name, you may leave the second parameter off.

Further parameters can be given to define attributes of the exported symbol. These parameters, like the second, are separated by white space. If further parameters are given, the external name must also be specified, even if it is the same as the internal name. The available attributes are:

- `resident` indicates that the exported name is to be kept resident by the system loader. This is an optimisation for frequently used symbols imported by name.
- `nodata` indicates that the exported symbol is a function which does not make use of any initialized data.
- `parm=NNN`, where `NNN` is an integer, sets the number of parameter words for the case in which the symbol is a call gate between 32−bit and 16−bit segments.
- An attribute which is just a number indicates that the symbol should be exported with an identifying number (ordinal), and gives the desired number.

For example:

```
export  myfunc
export  myfunc TheRealMoreFormalLookingFunctionName
export  myfunc myfunc 1234  ; export by ordinal
export  myfunc myfunc resident parm=23 nodata
```

### 6.2.6 `..start`: Defining the Program Entry Point

`OMF` linkers require exactly one of the object files being linked to define the program entry point, where execution will begin when the program is run. If the object file that defines the entry point is assembled using NASM, you specify the entry point by declaring the special symbol `..start` at the point where you wish execution to begin.

### 6.2.7 `obj` Extensions to the `EXTERN` Directive

If you declare an external symbol with the directive

```
extern  foo
```

then references such as `mov ax,foo` will give you the offset of `foo` from its preferred segment base (as specified in whichever module `foo` is actually defined in). So to access the contents of `foo` you will usually need to do something like

```
        mov     ax,seg foo      ; get preferred segment base
        mov     es,ax           ; move it into ES
        mov     ax,[es:foo]     ; and use offset ‘foo’ from it
```

This is a little unwieldy, particularly if you know that an external is going to be accessible from a given segment or group, say `dgroup`. So if `DS` already contained `dgroup`, you could simply code

```
        mov     ax,[foo wrt dgroup]
```

However, having to type this every time you want to access `foo` can be a pain; so NASM allows you to declare `foo` in the alternative form

```
extern  foo:wrt dgroup
```

This form causes NASM to pretend that the preferred segment base of `foo` is in fact `dgroup`; so the expression `seg foo` will now return `dgroup`, and the expression `foo` is equivalent to `foo wrt dgroup`.

This default−`WRT` mechanism can be used to make externals appear to be relative to any group or segment in your program. It can also be applied to common variables: see section 6.2.8.

### 6.2.8 `obj` Extensions to the `COMMON` Directive

The `obj` format allows common variables to be either near or far; NASM allows you to specify which your variables should be by the use of the syntax

```
common  nearvar 2:near   ; ‘nearvar’ is a near common
common  farvar  10:far   ; and ‘farvar’ is far
```

Far common variables may be greater in size than 64Kb, and so the OMF specification says that they are declared as a number of *elements* of a given size. So a 10−byte far common variable could be declared as ten one−byte elements, five two−byte elements, two five−byte elements or one ten−byte element.

Some `OMF` linkers require the element size, as well as the variable size, to match when resolving common variables declared in more than one module. Therefore NASM must allow you to specify the element size on your far common variables. This is done by the following syntax:

```
common  c_5by2  10:far 5        ; two five−byte elements
common  c_2by5  10:far 2        ; five two−byte elements
```

If no element size is specified, the default is 1. Also, the `FAR` keyword is not required when an element size is specified, since only far commons may have element sizes at all. So the above declarations could equivalently be

```
common  c_5by2  10:5            ; two five−byte elements
common  c_2by5  10:2            ; five two−byte elements
```

In addition to these extensions, the `COMMON` directive in `obj` also supports default−`WRT` specification like `EXTERN` does (explained in section 6.2.7). So you can also declare things like

```
common  foo     10:wrt dgroup
common  bar     16:far 2:wrt data
common  baz     24:wrt data:6
```

## 6.3 `win32`: Microsoft Win32 Object Files

The `win32` output format generates Microsoft Win32 object files, suitable for passing to Microsoft linkers such as Visual C++. Note that Borland Win32 compilers do not use this format, but use `obj` instead (see section 6.2).

`win32` provides a default output file−name extension of `.obj`.

Note that although Microsoft say that Win32 object files follow the `COFF` (Common Object File Format) standard, the object files produced by Microsoft Win32 compilers are not compatible with COFF linkers such as DJGPP's, and vice versa. This is due to a difference of opinion over the precise semantics of PC−relative relocations. To produce COFF files suitable for DJGPP, use NASM's `coff` output format; conversely, the `coff` format does not produce object files that Win32 linkers can generate correct output from.

### 6.3.1 `win32` Extensions to the `SECTION` Directive

Like the `obj` format, `win32` allows you to specify additional information on the `SECTION` directive line, to control the type and properties of sections you declare. Section types and properties are generated automatically by NASM for the standard section names `.text`, `.data` and `.bss`, but may still be overridden by these qualifiers.

The available qualifiers are:

- `code`, or equivalently `text`, defines the section to be a code section. This marks the section as readable and executable, but not writable, and also indicates to the linker that the type of the section is code.
- `data` and `bss` define the section to be a data section, analogously to `code`. Data sections are marked as readable and writable, but not executable. `data` declares an initialized data section, whereas `bss` declares an uninitialized data section.
- `rdata` declares an initialized data section that is readable but not writable. Microsoft compilers use this section to place constants in it.
- `info` defines the section to be an informational section, which is not included in the executable file by the linker, but may (for example) pass information *to* the linker. For example, declaring an `info`–type section called `.drectve` causes the linker to interpret the contents of the section as command−line options.
- `align=`, used with a trailing number as in `obj`, gives the alignment requirements of the section. The maximum you may specify is 64: the Win32 object file format contains no means to request a greater section alignment than this. If alignment is not explicitly specified, the defaults are 16−byte alignment for code sections, 8−byte alignment for rdata sections and 4−byte alignment for data (and BSS) sections. Informational sections get a default alignment of 1 byte (no alignment), though the value does not matter.

The defaults assumed by NASM if you do not specify the above qualifiers are:

```
section .text    code  align=16
section .data    data  align=4
section .rdata   rdata align=8
section .bss     bss   align=4
```

Any other section name is treated by default like `.text`.

## 6.4 `win64`: Microsoft Win64 Object Files

The `win64` output format generates Microsoft Win64 object files, which is nearly 100% indentical to the `win32` object format (section 6.3) with the exception that it is meant to target 64−bit code and the x86−64 platform altogether. This object file is used exactly the same as the `win32` object format (section 6.3), in NASM, with regard to this exception.

## 6.5 `coff`: Common Object File Format

The `coff` output type produces `COFF` object files suitable for linking with the DJGPP linker.

`coff` provides a default output file−name extension of `.o`.

The `coff` format supports the same extensions to the `SECTION` directive as `win32` does, except that the `align` qualifier and the `info` section type are not supported.

## 6.6 `macho`: Mach Object File Format

The `macho` output type produces `Mach−O` object files suitable for linking with the Mac OSX linker.

`macho` provides a default output file−name extension of `.o`.

## 6.7 `elf, elf32, and elf64`: Executable and Linkable Format Object Files

The `elf32` and `elf64` output formats generate `ELF32 and ELF64` (Executable and Linkable Format) object files, as used by Linux as well as Unix System V, including Solaris x86, UnixWare and SCO Unix. `elf` provides a default output file−name extension of `.o`. `elf` is a synonym for `elf32`.

### 6.7.1 `elf` Extensions to the `SECTION` Directive

Like the `obj` format, `elf` allows you to specify additional information on the `SECTION` directive line, to control the type and properties of sections you declare. Section types and properties are generated automatically by NASM for the standard section names `.text`, `.data` and `.bss`, but may still be overridden by these qualifiers.

The available qualifiers are:

- `alloc` defines the section to be one which is loaded into memory when the program is run. `noalloc` defines it to be one which is not, such as an informational or comment section.
- `exec` defines the section to be one which should have execute permission when the program is run. `noexec` defines it as one which should not.
- `write` defines the section to be one which should be writable when the program is run. `nowrite` defines it as one which should not.
- `progbits` defines the section to be one with explicit contents stored in the object file: an ordinary code or data section, for example, `nobits` defines the section to be one with no explicit contents given, such as a BSS section.
- `align=`, used with a trailing number as in `obj`, gives the alignment requirements of the section.

The defaults assumed by NASM if you do not specify the above qualifiers are:

```
section .text    progbits  alloc  exec    nowrite  align=16
section .rodata  progbits  alloc  noexec  nowrite  align=4
section .data    progbits  alloc  noexec  write    align=4
section .bss     nobits    alloc  noexec  write    align=4
section other    progbits  alloc  noexec  nowrite  align=1
```

(Any section name other than `.text`, `.rodata`, `.data` and `.bss` is treated by default like `other` in the above code.)

### 6.7.2 Position−Independent Code: `elf` Special Symbols and `WRT`

The `ELF` specification contains enough features to allow position−independent code (PIC) to be written, which makes ELF shared libraries very flexible. However, it also means NASM has to be able to generate a variety of strange relocation types in ELF object files, if it is to be an assembler which can write PIC.

Since `ELF` does not support segment−base references, the `WRT` operator is not used for its normal purpose; therefore NASM's `elf` output format makes use of `WRT` for a different purpose, namely the PIC−specific relocation types.

`elf` defines five special symbols which you can use as the right−hand side of the `WRT` operator to obtain PIC relocation types. They are `..gotpc`, `..gotoff`, `..got`, `..plt` and `..sym`. Their functions are summarized here:

- Referring to the symbol marking the global offset table base using `wrt ..gotpc` will end up giving the distance from the beginning of the current section to the global offset table. (`_GLOBAL_OFFSET_TABLE_` is the standard symbol name used to refer to the GOT.) So you would then need to add `$$` to the result to get the real address of the GOT.
- Referring to a location in one of your own sections using `wrt ..gotoff` will give the distance from the beginning of the GOT to the specified location, so that adding on the address of the GOT would give the real address of the location you wanted.
- Referring to an external or global symbol using `wrt ..got` causes the linker to build an entry *in* the GOT containing the address of the symbol, and the reference gives the distance from the beginning of the GOT to the entry; so you can add on the address of the GOT, load from the resulting address, and end up with the address of the symbol.
- Referring to a procedure name using `wrt ..plt` causes the linker to build a procedure linkage table entry for the symbol, and the reference gives the address of the PLT entry. You can only use this in contexts which would generate a PC−relative relocation normally (i.e. as the destination for `CALL` or `JMP`), since ELF contains no relocation type to refer to PLT entries absolutely.
- Referring to a symbol name using `wrt ..sym` causes NASM to write an ordinary relocation, but instead of making the relocation relative to the start of the section and then adding on the offset to the symbol, it will write a relocation record aimed directly at the symbol in question. The distinction is a necessary one due to a peculiarity of the dynamic linker.

A fuller explanation of how to use these relocation types to write shared libraries entirely in NASM is given in section 8.2.

### 6.7.3 `elf` Extensions to the `GLOBAL` Directive

`ELF` object files can contain more information about a global symbol than just its address: they can contain the size of the symbol and its type as well. These are not merely debugger conveniences, but are actually necessary when the program being written is a shared library. NASM therefore supports some extensions to the `GLOBAL` directive, allowing you to specify these features.

You can specify whether a global variable is a function or a data object by suffixing the name with a colon and the word `function` or `data`. (`object` is a synonym for `data`.) For example:

```
global   hashlookup:function, hashtable:data
```

exports the global symbol `hashlookup` as a function and `hashtable` as a data object.

Optionally, you can control the ELF visibility of the symbol. Just add one of the visibility keywords: `default`, `internal`, `hidden`, or `protected`. The default is `default` of course. For example, to make `hashlookup` hidden:

```
global   hashlookup:function hidden
```

You can also specify the size of the data associated with the symbol, as a numeric expression (which may involve labels, and even forward references) after the type specifier. Like this:

```
global  hashtable:data (hashtable.end - hashtable)

hashtable:
        db this,that,theother  ; some data here
.end:
```

This makes NASM automatically calculate the length of the table and place that information into the `ELF` symbol table.

Declaring the type and size of global symbols is necessary when writing shared library code. For more information, see section 8.2.4.

### 6.7.4 `elf` Extensions to the `COMMON` Directive

`ELF` also allows you to specify alignment requirements on common variables. This is done by putting a number (which must be a power of two) after the name and size of the common variable, separated (as usual) by a colon. For example, an array of doublewords would benefit from 4−byte alignment:

```
common  dwordarray 128:4
```

This declares the total size of the array to be 128 bytes, and requires that it be aligned on a 4−byte boundary.

### 6.7.5 16−bit code and ELF

The `ELF32` specification doesn't provide relocations for 8− and 16−bit values, but the GNU `ld` linker adds these as an extension. NASM can generate GNU−compatible relocations, to allow 16−bit code to be linked as ELF using GNU `ld`. If NASM is used with the `-w+gnu-elf-extensions` option, a warning is issued when one of these relocations is generated.

## 6.8 `aout`: Linux `a.out` Object Files

The `aout` format generates `a.out` object files, in the form used by early Linux systems (current Linux systems use ELF, see section 6.7.) These differ from other `a.out` object files in that the magic number in the first four bytes of the file is different; also, some implementations of `a.out`, for example NetBSD's, support position−independent code, which Linux's implementation does not.

`a.out` provides a default output file−name extension of `.o`.

`a.out` is a very simple object format. It supports no special directives, no special symbols, no use of `SEG` or `WRT`, and no extensions to any standard directives. It supports only the three standard section names `.text`, `.data` and `.bss`.

## 6.9 `aoutb`: NetBSD/FreeBSD/OpenBSD `a.out` Object Files

The `aoutb` format generates `a.out` object files, in the form used by the various free `BSD Unix` clones, `NetBSD`, `FreeBSD` and `OpenBSD`. For simple object files, this object format is exactly the same as `aout` except for the magic number in the first four bytes of the file. However, the `aoutb` format supports position−independent code in the same way as the `elf` format, so you can use it to write `BSD` shared libraries.

`aoutb` provides a default output file−name extension of `.o`.

`aoutb` supports no special directives, no special symbols, and only the three standard section names `.text`, `.data` and `.bss`. However, it also supports the same use of `WRT` as `elf` does, to provide position−independent code relocation types. See section 6.7.2 for full documentation of this feature.

`aoutb` also supports the same extensions to the `GLOBAL` directive as `elf` does: see section 6.7.3 for documentation of this.

## 6.10 `as86`: Minix/Linux `as86` Object Files

The Minix/Linux 16−bit assembler `as86` has its own non−standard object file format. Although its companion linker `ld86` produces something close to ordinary `a.out` binaries as output, the object file format used to communicate between `as86` and `ld86` is not itself `a.out`.

NASM supports this format, just in case it is useful, as `as86`. `as86` provides a default output file−name extension of `.o`.

`as86` is a very simple object format (from the NASM user’s point of view). It supports no special directives, no special symbols, no use of `SEG` or `WRT`, and no extensions to any standard directives. It supports only the three standard section names `.text`, `.data` and `.bss`.

## 6.11 `rdf`: Relocatable Dynamic Object File Format

The `rdf` output format produces `RDOFF` object files. `RDOFF` (Relocatable Dynamic Object File Format) is a home−grown object−file format, designed alongside NASM itself and reflecting in its file format the internal structure of the assembler.

`RDOFF` is not used by any well−known operating systems. Those writing their own systems, however, may well wish to use `RDOFF` as their object format, on the grounds that it is designed primarily for simplicity and contains very little file−header bureaucracy.

The Unix NASM archive, and the DOS archive which includes sources, both contain an `rdoff` subdirectory holding a set of RDOFF utilities: an RDF linker, an `RDF` static−library manager, an RDF file dump utility, and a program which will load and execute an RDF executable under Linux.

`rdf` supports only the standard section names `.text`, `.data` and `.bss`.

### 6.11.1 Requiring a Library: The `LIBRARY` Directive

`RDOFF` contains a mechanism for an object file to demand a given library to be linked to the module, either at load time or run time. This is done by the `LIBRARY` directive, which takes one argument which is the name of the module:

```
library  mylib.rdl
```

### 6.11.2 Specifying a Module Name: The `MODULE` Directive

Special `RDOFF` header record is used to store the name of the module. It can be used, for example, by run−time loader to perform dynamic linking. `MODULE` directive takes one argument which is the name of current module:

```
module  mymodname
```

Note that when you statically link modules and tell linker to strip the symbols from output file, all module names will be stripped too. To avoid it, you should start module names with `$`, like:

```
module  $kernel.core
```

### 6.11.3 `rdf` Extensions to the `GLOBAL` directive

`RDOFF` global symbols can contain additional information needed by the static linker. You can mark a global symbol as exported, thus telling the linker do not strip it from target executable or library file. Like in `ELF`, you can also specify whether an exported symbol is a procedure (function) or data object.

Suffixing the name with a colon and the word `export` you make the symbol exported:

```
global  sys_open:export
```

To specify that exported symbol is a procedure (function), you add the word `proc` or `function` after declaration:

```
global  sys_open:export proc
```

Similarly, to specify exported data object, add the word `data` or `object` to the directive:

```
global  kernel_ticks:export data
```

### 6.11.4 `rdf` Extensions to the `EXTERN` directive

By default the `EXTERN` directive in `RDOFF` declares a "pure external" symbol (i.e. the static linker will complain if such a symbol is not resolved). To declare an "imported" symbol, which must be resolved later during a dynamic linking phase, `RDOFF` offers an additional `import` modifier. As in `GLOBAL`, you can also specify whether an imported symbol is a procedure (function) or data object. For example:

```
library $libc
extern  _open:import
extern  _printf:import proc
extern  _errno:import data
```

Here the directive `LIBRARY` is also included, which gives the dynamic linker a hint as to where to find requested symbols.

## 6.12 `dbg`: Debugging Format

The `dbg` output format is not built into NASM in the default configuration. If you are building your own NASM executable from the sources, you can define `OF_DBG` in `outform.h` or on the compiler command line, and obtain the `dbg` output format.

The `dbg` format does not output an object file as such; instead, it outputs a text file which contains a complete list of all the transactions between the main body of NASM and the output−format back end module. It is primarily intended to aid people who want to write their own output drivers, so that they can get a clearer idea of the various requests the main program makes of the output driver, and in what order they happen.

For simple files, one can easily use the `dbg` format like this:

```
nasm −f dbg filename.asm
```

which will generate a diagnostic file called `filename.dbg`. However, this will not work well on files which were designed for a different object format, because each object format defines its own macros (usually user−level forms of directives), and those macros will not be defined in the `dbg` format. Therefore it can be useful to run NASM twice, in order to do the preprocessing with the native object format selected:

```
nasm −e −f rdf −o rdfprog.i rdfprog.asm
nasm −a −f dbg rdfprog.i
```

This preprocesses `rdfprog.asm` into `rdfprog.i`, keeping the `rdf` object format selected in order to make sure RDF special directives are converted into primitive form correctly. Then the preprocessed source is fed through the `dbg` format to generate the final diagnostic output.

This workaround will still typically not work for programs intended for `obj` format, because the `obj` `SEGMENT` and `GROUP` directives have side effects of defining the segment and group names as symbols; `dbg` will not do this, so the program will not assemble. You will have to work around that by defining the symbols yourself (using `EXTERN`, for example) if you really need to get a `dbg` trace of an `obj`–specific source file.

`dbg` accepts any section name and any directives at all, and logs them all to its output file.

# Chapter 7: Writing 16−bit Code (DOS, Windows 3/3.1)

This chapter attempts to cover some of the common issues encountered when writing 16−bit code to run under `MS−DOS` or `Windows 3.x`. It covers how to link programs to produce `.EXE` or `.COM` files, how to write `.SYS` device drivers, and how to interface assembly language code with 16−bit C compilers and with Borland Pascal.

## 7.1 Producing `.EXE` Files

Any large program written under DOS needs to be built as a `.EXE` file: only `.EXE` files have the necessary internal structure required to span more than one 64K segment. Windows programs, also, have to be built as `.EXE` files, since Windows does not support the `.COM` format.

In general, you generate `.EXE` files by using the `obj` output format to produce one or more `.OBJ` files, and then linking them together using a linker. However, NASM also supports the direct generation of simple DOS `.EXE` files using the `bin` output format (by using `DB` and `DW` to construct the `.EXE` file header), and a macro package is supplied to do this. Thanks to Yann Guidon for contributing the code for this.

NASM may also support `.EXE` natively as another output format in future releases.

### 7.1.1 Using the `obj` Format To Generate `.EXE` Files

This section describes the usual method of generating `.EXE` files by linking `.OBJ` files together.

Most 16−bit programming language packages come with a suitable linker; if you have none of these, there is a free linker called VAL, available in `LZH` archive format from `x2ftp.oulu.fi`. An LZH archiver can be found at `ftp.simtel.net`. There is another ‘free’ linker (though this one doesn’t come with sources) called FREELINK, available from `www.pcorner.com`. A third, `djlink`, written by DJ Delorie, is available at `www.delorie.com`. A fourth linker, `ALINK`, written by Anthony A.J. Williams, is available at `alink.sourceforge.net`.

When linking several `.OBJ` files into a `.EXE` file, you should ensure that exactly one of them has a start point defined (using the `..start` special symbol defined by the `obj` format: see section 6.2.6). If no module defines a start point, the linker will not know what value to give the entry−point field in the output file header; if more than one defines a start point, the linker will not know *which* value to use.

An example of a NASM source file which can be assembled to a `.OBJ` file and linked on its own to a `.EXE` is given here. It demonstrates the basic principles of defining a stack, initialising the segment registers, and declaring a start point. This file is also provided in the `test` subdirectory of the NASM archives, under the name `objexe.asm`.

```
segment code

..start:
        mov     ax,data
        mov     ds,ax
        mov     ax,stack
        mov     ss,ax
        mov     sp,stacktop
```

This initial piece of code sets up `DS` to point to the data segment, and initializes `SS` and `SP` to point to the top of the provided stack. Notice that interrupts are implicitly disabled for one instruction after a move into `SS`, precisely for this situation, so that there's no chance of an interrupt occurring between the loads of `SS` and `SP` and not having a stack to execute on.

Note also that the special symbol `..start` is defined at the beginning of this code, which means that will be the entry point into the resulting executable file.

```
        mov     dx,hello
        mov     ah,9
        int     0x21
```

The above is the main program: load `DS:DX` with a pointer to the greeting message (`hello` is implicitly relative to the segment `data`, which was loaded into `DS` in the setup code, so the full pointer is valid), and call the DOS print−string function.

```
        mov     ax,0x4c00
        int     0x21
```

This terminates the program using another DOS system call.

```
segment data

hello:  db      'hello, world', 13, 10, '$'
```

The data segment contains the string we want to display.

```
segment stack stack
        resb 64
stacktop:
```

The above code declares a stack segment containing 64 bytes of uninitialized stack space, and points `stacktop` at the top of it. The directive `segment stack stack` defines a segment *called* `stack`, and also of *type* `STACK`. The latter is not necessary to the correct running of the program, but linkers are likely to issue warnings or errors if your program has no segment of type `STACK`.

The above file, when assembled into a `.OBJ` file, will link on its own to a valid `.EXE` file, which when run will print 'hello, world' and then exit.

### 7.1.2 Using the `bin` Format To Generate `.EXE` Files

The `.EXE` file format is simple enough that it's possible to build a `.EXE` file by writing a pure−binary program and sticking a 32−byte header on the front. This header is simple enough that it can be generated using `DB` and `DW` commands by NASM itself, so that you can use the `bin` output format to directly generate `.EXE` files.

Included in the NASM archives, in the `misc` subdirectory, is a file `exebin.mac` of macros. It defines three macros: `EXE_begin`, `EXE_stack` and `EXE_end`.

To produce a `.EXE` file using this method, you should start by using `%include` to load the `exebin.mac` macro package into your source file. You should then issue the `EXE_begin` macro call (which takes no arguments) to generate the file header data. Then write code as normal for the `bin` format – you can use all three standard sections `.text`, `.data` and `.bss`. At the end of the file you should call the `EXE_end` macro (again, no arguments), which defines some symbols to mark section sizes, and these symbols are referred to in the header code generated by `EXE_begin`.

In this model, the code you end up writing starts at `0x100`, just like a `.COM` file – in fact, if you strip off the 32−byte header from the resulting `.EXE` file, you will have a valid `.COM` program. All the segment bases are the same, so you are limited to a 64K program, again just like a `.COM` file.

Note that an `ORG` directive is issued by the `EXE_begin` macro, so you should not explicitly issue one of your own.

You can't directly refer to your segment base value, unfortunately, since this would require a relocation in the header, and things would get a lot more complicated. So you should get your segment base by copying it out of `CS` instead.

On entry to your `.EXE` file, `SS:SP` are already set up to point to the top of a 2Kb stack. You can adjust the default stack size of 2Kb by calling the `EXE_stack` macro. For example, to change the stack size of your program to 64 bytes, you would call `EXE_stack 64`.

A sample program which generates a `.EXE` file in this way is given in the `test` subdirectory of the NASM archive, as `binexe.asm`.

## 7.2 Producing `.COM` Files

While large DOS programs must be written as `.EXE` files, small ones are often better written as `.COM` files. `.COM` files are pure binary, and therefore most easily produced using the `bin` output format.

### 7.2.1 Using the `bin` Format To Generate `.COM` Files

`.COM` files expect to be loaded at offset `100h` into their segment (though the segment may change). Execution then begins at `100h`, i.e. right at the start of the program. So to write a `.COM` program, you would create a source file looking like

```
        org 100h

section .text

start:
        ; put your code here

section .data

        ; put data items here

section .bss

        ; put uninitialized data here
```

The `bin` format puts the `.text` section first in the file, so you can declare data or BSS items before beginning to write code if you want to and the code will still end up at the front of the file where it belongs.

The BSS (uninitialized data) section does not take up space in the `.COM` file itself: instead, addresses of BSS items are resolved to point at space beyond the end of the file, on the grounds that this will be free memory when the program is run. Therefore you should not rely on your BSS being initialized to all zeros when you run.

To assemble the above program, you should use a command line like

```
nasm myprog.asm −fbin −o myprog.com
```

The `bin` format would produce a file called `myprog` if no explicit output file name were specified, so you have to override it and give the desired file name.

### 7.2.2 Using the `obj` Format To Generate `.COM` Files

If you are writing a `.COM` program as more than one module, you may wish to assemble several `.OBJ` files and link them together into a `.COM` program. You can do this, provided you have a linker capable of outputting `.COM` files directly (TLINK does this), or alternatively a converter program such as `EXE2BIN` to transform the `.EXE` file output from the linker into a `.COM` file.

If you do this, you need to take care of several things:

- The first object file containing code should start its code segment with a line like `RESB 100h`. This is to ensure that the code begins at offset `100h` relative to the beginning of the code segment, so that the linker or converter program does not have to adjust address references within the file when generating the `.COM` file. Other assemblers use an `ORG` directive for this purpose, but `ORG` in NASM is a format−specific directive to the `bin` output format, and does not mean the same thing as it does in MASM−compatible assemblers.
- You don’t need to define a stack segment.
- All your segments should be in the same group, so that every time your code or data references a symbol offset, all offsets are relative to the same segment base. This is because, when a `.COM` file is loaded, all the segment registers contain the same value.

## 7.3 Producing `.SYS` Files

MS−DOS device drivers – `.SYS` files – are pure binary files, similar to `.COM` files, except that they start at origin zero rather than `100h`. Therefore, if you are writing a device driver using the `bin` format, you do not need the `ORG` directive, since the default origin for `bin` is zero. Similarly, if you are using `obj`, you do not need the `RESB 100h` at the start of your code segment.

`.SYS` files start with a header structure, containing pointers to the various routines inside the driver which do the work. This structure should be defined at the start of the code segment, even though it is not actually code.

For more information on the format of `.SYS` files, and the data which has to go in the header structure, a list of books is given in the Frequently Asked Questions list for the newsgroup `comp.os.msdos.programmer`.

## 7.4 Interfacing to 16−bit C Programs

This section covers the basics of writing assembly routines that call, or are called from, C programs. To do this, you would typically write an assembly module as a `.OBJ` file, and link it with your C modules to produce a mixed−language program.

### 7.4.1 External Symbol Names

C compilers have the convention that the names of all global symbols (functions or data) they define are formed by prefixing an underscore to the name as it appears in the C program. So, for example, the function a C programmer thinks of as `printf` appears to an assembly language programmer as `_printf`. This means that in your assembly programs, you can define symbols without a leading underscore, and not have to worry about name clashes with C symbols.

If you find the underscores inconvenient, you can define macros to replace the `GLOBAL` and `EXTERN` directives as follows:

```
%macro  cglobal 1

  global  _%1
  %define %1 _%1
```

```
%endmacro

%macro  cextern 1

  extern  _%1
  %define %1 _%1

%endmacro
```

(These forms of the macros only take one argument at a time; a `%rep` construct could solve this.)

If you then declare an external like this:

```
cextern printf
```

then the macro will expand it as

```
extern  _printf
%define printf _printf
```

Thereafter, you can reference `printf` as if it was a symbol, and the preprocessor will put the leading underscore on where necessary.

The `cglobal` macro works similarly. You must use `cglobal` before defining the symbol in question, but you would have had to do that anyway if you used `GLOBAL`.

Also see section 2.1.22.

### 7.4.2 Memory Models

NASM contains no mechanism to support the various C memory models directly; you have to keep track yourself of which one you are writing for. This means you have to keep track of the following things:

- In models using a single code segment (tiny, small and compact), functions are near. This means that function pointers, when stored in data segments or pushed on the stack as function arguments, are 16 bits long and contain only an offset field (the `CS` register never changes its value, and always gives the segment part of the full function address), and that functions are called using ordinary near `CALL` instructions and return using `RETN` (which, in NASM, is synonymous with `RET` anyway). This means both that you should write your own routines to return with `RETN`, and that you should call external C routines with near `CALL` instructions.
- In models using more than one code segment (medium, large and huge), functions are far. This means that function pointers are 32 bits long (consisting of a 16−bit offset followed by a 16−bit segment), and that functions are called using `CALL FAR` (or `CALL seg:offset`) and return using `RETF`. Again, you should therefore write your own routines to return with `RETF` and use `CALL FAR` to call external routines.
- In models using a single data segment (tiny, small and medium), data pointers are 16 bits long, containing only an offset field (the `DS` register doesn't change its value, and always gives the segment part of the full data item address).
- In models using more than one data segment (compact, large and huge), data pointers are 32 bits long, consisting of a 16−bit offset followed by a 16−bit segment. You should still be careful not to modify `DS` in your routines without restoring it afterwards, but `ES` is free for you to use to access the contents of 32−bit data pointers you are passed.
- The huge memory model allows single data items to exceed 64K in size. In all other memory models, you can access the whole of a data item just by doing arithmetic on the offset field of the

pointer you are given, whether a segment field is present or not; in huge model, you have to be more careful of your pointer arithmetic.

- In most memory models, there is a *default* data segment, whose segment address is kept in `DS` throughout the program. This data segment is typically the same segment as the stack, kept in `SS`, so that functions' local variables (which are stored on the stack) and global data items can both be accessed easily without changing `DS`. Particularly large data items are typically stored in other segments. However, some memory models (though not the standard ones, usually) allow the assumption that `SS` and `DS` hold the same value to be removed. Be careful about functions' local variables in this latter case.

In models with a single code segment, the segment is called `_TEXT`, so your code segment must also go by this name in order to be linked into the same place as the main code segment. In models with a single data segment, or with a default data segment, it is called `_DATA`.

### 7.4.3 Function Definitions and Function Calls

The C calling convention in 16−bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function's parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
- The caller then executes a `CALL` instruction to pass control to the callee. This `CALL` is either near or far depending on the memory model.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `SP` in `BP` so as to be able to use `BP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `BP` must be preserved by any C function. Hence the callee, if it is going to set up `BP` as a *frame pointer*, must push the previous value first.
- The callee may then access its parameters relative to `BP`. The word at `[BP]` holds the previous value of `BP` as it was pushed; the next word, at `[BP+2]`, holds the offset part of the return address, pushed implicitly by `CALL`. In a small−model (near) function, the parameters start after that, at `[BP+4]`; in a large−model (far) function, the segment part of the return address lives at `[BP+4]`, and the parameters begin at `[BP+6]`. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from `BP`; the others follow, at successively greater offsets. Thus, in a function such as `printf` which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
- The callee may also wish to decrease `SP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `BP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `DX:AX` depending on the size of the value. Floating−point results are sometimes (depending on the compiler) returned in `ST0`.
- Once the callee has finished processing, it restores `SP` from `BP` if it had allocated local stack space, then pops the previous value of `BP`, and returns via `RETN` or `RETF` depending on memory model.
- When the caller regains control from the callee, the function parameters are still on the stack, so it typically adds an immediate constant to `SP` to remove them (instead of executing a number of slow `POP` instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the caller, which *knows* how many parameters it pushed, does the removing.

It is instructive to compare this calling convention with that for Pascal programs (described in section 7.5.1). Pascal has a simpler convention, since no functions have variable numbers of parameters. Therefore the callee knows how many parameters it should have been passed, and is able to deallocate them from the stack itself by passing an immediate argument to the `RET` or `RETF` instruction, so the caller does not have to do it. Also, the parameters are pushed in left−to−right order, not right−to−left, which means that a compiler can give better guarantees about sequence points without performance suffering.

Thus, you would define a function in C style in the following way. The following example is for small model:

```
global  _myfunc

_myfunc:
        push    bp
        mov     bp,sp
        sub     sp,0x40         ; 64 bytes of local stack space
        mov     bx,[bp+4]       ; first parameter to function

        ; some more code

        mov     sp,bp           ; undo "sub sp,0x40" above
        pop     bp
        ret
```

For a large−model function, you would replace `RET` by `RETF`, and look for the first parameter at `[BP+6]` instead of `[BP+4]`. Of course, if one of the parameters is a pointer, then the offsets of *subsequent* parameters will change depending on the memory model as well: far pointers take up four bytes on the stack when passed as a parameter, whereas near pointers take up two.

At the other end of the process, to call a C function from your assembly code, you would do something like this:

```
extern  _printf

      ; and then, further down...

      push    word [myint]        ; one of my integer variables
      push    word mystring       ; pointer into my data segment
      call    _printf
      add     sp,byte 4           ; ‘byte’ saves space

      ; then those data items...

segment _DATA

myint         dw    1234
mystring      db    ’This number −> %d <− should be 1234’,10,0
```

This piece of code is the small−model assembly equivalent of the C code

```
    int myint = 1234;
    printf("This number −> %d <− should be 1234\\n", myint);
```

In large model, the function−call code might look more like this. In this example, it is assumed that `DS` already holds the segment base of the segment `_DATA`. If not, you would have to initialize it first.

```
        push    word [myint]
        push    word seg mystring   ; Now push the segment, and...
        push    word mystring       ; ... offset of "mystring"
        call    far _printf
        add    sp,byte 6
```

The integer value still takes up one word on the stack, since large model does not affect the size of the `int` data type. The first argument (pushed last) to `printf`, however, is a data pointer, and therefore has to contain a segment and offset part. The segment should be stored second in memory, and therefore must be pushed first. (Of course, `PUSH DS` would have been a shorter instruction than `PUSH WORD SEG mystring`, if `DS` was set up as the above example assumed.) Then the actual call becomes a far call, since functions expect far calls in large model; and `SP` has to be increased by 6 rather than 4 afterwards to make up for the extra word of parameters.

### 7.4.4 Accessing Data Items

To get at the contents of C variables, or to declare variables which C can access, you need only declare the names as `GLOBAL` or `EXTERN`. (Again, the names require leading underscores, as stated in section 7.4.1.) Thus, a C variable declared as `int i` can be accessed from assembler as

```
extern _i

        mov ax,[_i]
```

And to declare your own integer variable which C programs can access as `extern int j`, you do this (making sure you are assembling in the `_DATA` segment, if necessary):

```
global  _j

_j      dw      0
```

To access a C array, you need to know the size of the components of the array. For example, `int` variables are two bytes long, so if a C program declares an array as `int a[10]`, you can access `a[3]` by coding `mov ax,[_a+6]`. (The byte offset 6 is obtained by multiplying the desired array index, 3, by the size of the array element, 2.) The sizes of the C base types in 16−bit compilers are: 1 for `char`, 2 for `short` and `int`, 4 for `long` and `float`, and 8 for `double`.

To access a C data structure, you need to know the offset from the base of the structure to the field you are interested in. You can either do this by converting the C structure definition into a NASM structure definition (using `STRUC`), or by calculating the one offset and using just that.

To do either of these, you should read your C compiler’s manual to find out how it organizes data structures. NASM gives no special alignment to structure members in its own `STRUC` macro, so you have to specify alignment yourself if the C compiler generates it. Typically, you might find that a structure like

```
struct @\{
    char c;
    int i;
@\} foo;
```

might be four bytes long rather than three, since the `int` field would be aligned to a two−byte boundary. However, this sort of feature tends to be a configurable option in the C compiler, either using command−line options or `#pragma` lines, so you have to find out how your own compiler does it.

### 7.4.5 `c16.mac`: Helper Macros for the 16−bit C Interface

Included in the NASM archives, in the `misc` directory, is a file `c16.mac` of macros. It defines three macros: `proc`, `arg` and `endproc`. These are intended to be used for C−style procedure definitions, and they automate a lot of the work involved in keeping track of the calling convention.

(An alternative, TASM compatible form of `arg` is also now built into NASM's preprocessor. See section 4.9 for details.)

An example of an assembly function using the macro set is given here:

```
proc    _nearproc

%$i     arg
%$j     arg
        mov     ax,[bp + %$i]
        mov     bx,[bp + %$j]
        add     ax,[bx]

endproc
```

This defines `_nearproc` to be a procedure taking two arguments, the first (`i`) an integer and the second (`j`) a pointer to an integer. It returns `i + *j`.

Note that the `arg` macro has an `EQU` as the first line of its expansion, and since the label before the macro call gets prepended to the first line of the expanded macro, the `EQU` works, defining `%$i` to be an offset from `BP`. A context−local variable is used, local to the context pushed by the `proc` macro and popped by the `endproc` macro, so that the same argument name can be used in later procedures. Of course, you don't *have* to do that.

The macro set produces code for near functions (tiny, small and compact−model code) by default. You can have it generate far functions (medium, large and huge−model code) by means of coding `%define FARCODE`. This changes the kind of return instruction generated by `endproc`, and also changes the starting point for the argument offsets. The macro set contains no intrinsic dependency on whether data pointers are far or not.

`arg` can take an optional parameter, giving the size of the argument. If no size is given, 2 is assumed, since it is likely that many function parameters will be of type `int`.

The large−model equivalent of the above function would look like this:

```
%define FARCODE

proc    _farproc

%$i     arg
%$j     arg     4
        mov     ax,[bp + %$i]
        mov     bx,[bp + %$j]
        mov     es,[bp + %$j + 2]
        add     ax,[bx]

endproc
```

This makes use of the argument to the `arg` macro to define a parameter of size 4, because `j` is now a far pointer. When we load from `j`, we must load a segment and an offset.

## 7.5 Interfacing to Borland Pascal Programs

Interfacing to Borland Pascal programs is similar in concept to interfacing to 16−bit C programs. The differences are:

- The leading underscore required for interfacing to C programs is not required for Pascal.
- The memory model is always large: functions are far, data pointers are far, and no data item can be more than 64K long. (Actually, some functions are near, but only those functions that are local to a Pascal unit and never called from outside it. All assembly functions that Pascal calls, and all Pascal functions that assembly routines are able to call, are far.) However, all static data declared in a Pascal program goes into the default data segment, which is the one whose segment address will be in `DS` when control is passed to your assembly code. The only things that do not live in the default data segment are local variables (they live in the stack segment) and dynamically allocated variables. All data *pointers*, however, are far.
- The function calling convention is different – described below.
- Some data types, such as strings, are stored differently.
- There are restrictions on the segment names you are allowed to use – Borland Pascal will ignore code or data declared in a segment it doesn't like the name of. The restrictions are described below.

### 7.5.1 The Pascal Calling Convention

The 16−bit Pascal calling convention is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function's parameters on the stack, one after another, in normal order (left to right, so that the first argument specified to the function is pushed first).
- The caller then executes a far `CALL` instruction to pass control to the callee.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `SP` in `BP` so as to be able to use `BP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `BP` must be preserved by any function. Hence the callee, if it is going to set up `BP` as a frame pointer, must push the previous value first.
- The callee may then access its parameters relative to `BP`. The word at `[BP]` holds the previous value of `BP` as it was pushed. The next word, at `[BP+2]`, holds the offset part of the return address, and the next one at `[BP+4]` the segment part. The parameters begin at `[BP+6]`. The rightmost parameter of the function, since it was pushed last, is accessible at this offset from `BP`; the others follow, at successively greater offsets.
- The callee may also wish to decrease `SP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `BP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `DX:AX` depending on the size of the value. Floating−point results are returned in `ST0`. Results of type `Real` (Borland's own custom floating−point data type, not handled directly by the FPU) are returned in `DX:BX:AX`. To return a result of type `String`, the caller pushes a pointer to a temporary string before pushing the parameters, and the callee places the returned string value at that location. The pointer is not a parameter, and should not be removed from the stack by the `RETF` instruction.
- Once the callee has finished processing, it restores `SP` from `BP` if it had allocated local stack space, then pops the previous value of `BP`, and returns via `RETF`. It uses the form of `RETF` with

an immediate parameter, giving the number of bytes taken up by the parameters on the stack. This causes the parameters to be removed from the stack as a side effect of the return instruction.

- When the caller regains control from the callee, the function parameters have already been removed from the stack, so it needs to do nothing further.

Thus, you would define a function in Pascal style, taking two `Integer`–type parameters, in the following way:

```
global  myfunc

myfunc: push    bp
        mov     bp,sp
        sub     sp,0x40         ; 64 bytes of local stack space
        mov     bx,[bp+8]       ; first parameter to function
        mov     bx,[bp+6]       ; second parameter to function

        ; some more code

        mov     sp,bp           ; undo "sub sp,0x40" above
        pop     bp
        retf    4               ; total size of params is 4
```

At the other end of the process, to call a Pascal function from your assembly code, you would do something like this:

```
extern  SomeFunc

       ; and then, further down...

       push   word seg mystring   ; Now push the segment, and...
       push   word mystring       ; ... offset of "mystring"
       push   word [myint]        ; one of my variables
       call   far SomeFunc
```

This is equivalent to the Pascal code

```
procedure SomeFunc(String: PChar; Int: Integer);
    SomeFunc(@@mystring, myint);
```

### 7.5.2 Borland Pascal Segment Name Restrictions

Since Borland Pascal’s internal unit file format is completely different from `OBJ`, it only makes a very sketchy job of actually reading and understanding the various information contained in a real `OBJ` file when it links that in. Therefore an object file intended to be linked to a Pascal program must obey a number of restrictions:

- Procedures and functions must be in a segment whose name is either `CODE`, `CSEG`, or something ending in `_TEXT`.
- initialized data must be in a segment whose name is either `CONST` or something ending in `_DATA`.
- Uninitialized data must be in a segment whose name is either `DATA`, `DSEG`, or something ending in `_BSS`.
- Any other segments in the object file are completely ignored. `GROUP` directives and segment attributes are also ignored.

### 7.5.3 Using `c16.mac` With Pascal Programs

The `c16.mac` macro package, described in section 7.4.5, can also be used to simplify writing functions to be called from Pascal programs, if you code `%define PASCAL`. This definition ensures that functions are far (it implies `FARCODE`), and also causes procedure return instructions to be generated with an operand.

Defining `PASCAL` does not change the code which calculates the argument offsets; you must declare your function's arguments in reverse order. For example:

```
%define PASCAL

proc    _pascalproc

%$j     arg 4
%$i     arg
        mov     ax,[bp + %$i]
        mov     bx,[bp + %$j]
        mov     es,[bp + %$j + 2]
        add     ax,[bx]

endproc
```

This defines the same routine, conceptually, as the example in section 7.4.5: it defines a function taking two arguments, an integer and a pointer to an integer, which returns the sum of the integer and the contents of the pointer. The only difference between this code and the large−model C version is that `PASCAL` is defined instead of `FARCODE`, and that the arguments are declared in reverse order.

# Chapter 8: Writing 32−bit Code (Unix, Win32, DJGPP)

This chapter attempts to cover some of the common issues involved when writing 32−bit code, to run under Win32 or Unix, or to be linked with C code generated by a Unix−style C compiler such as DJGPP. It covers how to write assembly code to interface with 32−bit C routines, and how to write position−independent code for shared libraries.

Almost all 32−bit code, and in particular all code running under `Win32`, `DJGPP` or any of the PC Unix variants, runs in *flat* memory model. This means that the segment registers and paging have already been set up to give you the same 32−bit 4Gb address space no matter what segment you work relative to, and that you should ignore all segment registers completely. When writing flat−model application code, you never need to use a segment override or modify any segment register, and the code−section addresses you pass to `CALL` and `JMP` live in the same address space as the data−section addresses you access your variables by and the stack−section addresses you access local variables and procedure parameters by. Every address is 32 bits long and contains only an offset part.

## 8.1 Interfacing to 32−bit C Programs

A lot of the discussion in section 7.4, about interfacing to 16−bit C programs, still applies when working in 32 bits. The absence of memory models or segmentation worries simplifies things a lot.

### 8.1.1 External Symbol Names

Most 32−bit C compilers share the convention used by 16−bit compilers, that the names of all global symbols (functions or data) they define are formed by prefixing an underscore to the name as it appears in the C program. However, not all of them do: the `ELF` specification states that C symbols do *not* have a leading underscore on their assembly−language names.

The older Linux `a.out` C compiler, all `Win32` compilers, `DJGPP`, and `NetBSD` and `FreeBSD`, all use the leading underscore; for these compilers, the macros `cextern` and `cglobal`, as given in section 7.4.1, will still work. For `ELF`, though, the leading underscore should not be used.

See also section 2.1.22.

### 8.1.2 Function Definitions and Function Calls

The C calling conventionThe C calling convention in 32−bit programs is as follows. In the following description, the words *caller* and *callee* are used to denote the function doing the calling and the function which gets called.

- The caller pushes the function’s parameters on the stack, one after another, in reverse order (right to left, so that the first argument specified to the function is pushed last).
- The caller then executes a near `CALL` instruction to pass control to the callee.
- The callee receives control, and typically (although this is not actually necessary, in functions which do not need to access their parameters) starts by saving the value of `ESP` in `EBP` so as to be able to use `EBP` as a base pointer to find its parameters on the stack. However, the caller was probably doing this too, so part of the calling convention states that `EBP` must be preserved by any C function. Hence the callee, if it is going to set up `EBP` as a frame pointer, must push the previous value first.

- The callee may then access its parameters relative to `EBP`. The doubleword at `[EBP]` holds the previous value of `EBP` as it was pushed; the next doubleword, at `[EBP+4]`, holds the return address, pushed implicitly by `CALL`. The parameters start after that, at `[EBP+8]`. The leftmost parameter of the function, since it was pushed last, is accessible at this offset from `EBP`; the others follow, at successively greater offsets. Thus, in a function such as `printf` which takes a variable number of parameters, the pushing of the parameters in reverse order means that the function knows where to find its first parameter, which tells it the number and type of the remaining ones.
- The callee may also wish to decrease `ESP` further, so as to allocate space on the stack for local variables, which will then be accessible at negative offsets from `EBP`.
- The callee, if it wishes to return a value to the caller, should leave the value in `AL`, `AX` or `EAX` depending on the size of the value. Floating−point results are typically returned in `ST0`.
- Once the callee has finished processing, it restores `ESP` from `EBP` if it had allocated local stack space, then pops the previous value of `EBP`, and returns via `RET` (equivalently, `RETN`).
- When the caller regains control from the callee, the function parameters are still on the stack, so it typically adds an immediate constant to `ESP` to remove them (instead of executing a number of slow `POP` instructions). Thus, if a function is accidentally called with the wrong number of parameters due to a prototype mismatch, the stack will still be returned to a sensible state since the caller, which *knows* how many parameters it pushed, does the removing.

There is an alternative calling convention used by Win32 programs for Windows API calls, and also for functions called *by* the Windows API such as window procedures: they follow what Microsoft calls the `__stdcall` convention. This is slightly closer to the Pascal convention, in that the callee clears the stack by passing a parameter to the `RET` instruction. However, the parameters are still pushed in right−to−left order.

Thus, you would define a function in C style in the following way:

```
global  _myfunc

_myfunc:
        push    ebp
        mov     ebp,esp
        sub     esp,0x40        ; 64 bytes of local stack space
        mov     ebx,[ebp+8]     ; first parameter to function

        ; some more code

        leave                   ; mov esp,ebp / pop ebp
        ret
```

At the other end of the process, to call a C function from your assembly code, you would do something like this:

```
extern  _printf

        ; and then, further down...

        push    dword [myint]   ; one of my integer variables
        push    dword mystring  ; pointer into my data segment
        call    _printf
        add     esp,byte 8      ; ‘byte’ saves space
        ; then those data items...

segment _DATA

myint       dd   1234
mystring    db   'This number -> %d <- should be 1234',10,0
```

This piece of code is the assembly equivalent of the C code

```
    int myint = 1234;
    printf("This number -> %d <- should be 1234\\n", myint);
```

### 8.1.3 Accessing Data Items

To get at the contents of C variables, or to declare variables which C can access, you need only declare the names as `GLOBAL` or `EXTERN`. (Again, the names require leading underscores, as stated in section 8.1.1.) Thus, a C variable declared as `int i` can be accessed from assembler as

```
          extern _i
          mov eax,[_i]
```

And to declare your own integer variable which C programs can access as `extern int j`, you do this (making sure you are assembling in the `_DATA` segment, if necessary):

```
          global _j
_j        dd 0
```

To access a C array, you need to know the size of the components of the array. For example, `int` variables are four bytes long, so if a C program declares an array as `int a[10]`, you can access `a[3]` by coding `mov ax,[_a+12]`. (The byte offset 12 is obtained by multiplying the desired array index, 3, by the size of the array element, 4.) The sizes of the C base types in 32−bit compilers are: 1 for `char`, 2 for `short`, 4 for `int`, `long` and `float`, and 8 for `double`. Pointers, being 32−bit addresses, are also 4 bytes long.

To access a C data structure, you need to know the offset from the base of the structure to the field you are interested in. You can either do this by converting the C structure definition into a NASM structure definition (using `STRUC`), or by calculating the one offset and using just that.

To do either of these, you should read your C compiler's manual to find out how it organizes data structures. NASM gives no special alignment to structure members in its own `STRUC` macro, so you have to specify alignment yourself if the C compiler generates it. Typically, you might find that a structure like

```
struct @\{
    char c;
    int i;
@\} foo;
```

might be eight bytes long rather than five, since the `int` field would be aligned to a four−byte boundary. However, this sort of feature is sometimes a configurable option in the C compiler, either using command−line options or `#pragma` lines, so you have to find out how your own compiler does it.

### 8.1.4 `c32.mac`: Helper Macros for the 32−bit C Interface

Included in the NASM archives, in the `misc` directory, is a file `c32.mac` of macros. It defines three macros: `proc`, `arg` and `endproc`. These are intended to be used for C−style procedure definitions, and they automate a lot of the work involved in keeping track of the calling convention.

An example of an assembly function using the macro set is given here:

```
proc    _proc32

%$i     arg
%$j     arg
        mov     eax,[ebp + %$i]
        mov     ebx,[ebp + %$j]
        add     eax,[ebx]

endproc
```

This defines `_proc32` to be a procedure taking two arguments, the first (`i`) an integer and the second (`j`) a pointer to an integer. It returns `i + *j`.

Note that the `arg` macro has an `EQU` as the first line of its expansion, and since the label before the macro call gets prepended to the first line of the expanded macro, the `EQU` works, defining `%$i` to be an offset from `BP`. A context−local variable is used, local to the context pushed by the `proc` macro and popped by the `endproc` macro, so that the same argument name can be used in later procedures. Of course, you don't *have* to do that.

`arg` can take an optional parameter, giving the size of the argument. If no size is given, 4 is assumed, since it is likely that many function parameters will be of type `int` or pointers.

## 8.2 Writing NetBSD/FreeBSD/OpenBSD and Linux/ELF Shared Libraries

`ELF` replaced the older `a.out` object file format under Linux because it contains support for position−independent code (PIC), which makes writing shared libraries much easier. NASM supports the `ELF` position−independent code features, so you can write Linux `ELF` shared libraries in NASM.

NetBSD, and its close cousins FreeBSD and OpenBSD, take a different approach by hacking PIC support into the `a.out` format. NASM supports this as the `aoutb` output format, so you can write BSD shared libraries in NASM too.

The operating system loads a PIC shared library by memory−mapping the library file at an arbitrarily chosen point in the address space of the running process. The contents of the library's code section must therefore not depend on where it is loaded in memory.

Therefore, you cannot get at your variables by writing code like this:

```
        mov     eax,[myvar]             ; WRONG
```

Instead, the linker provides an area of memory called the *global offset table*, or GOT; the GOT is situated at a constant distance from your library's code, so if you can find out where your library is loaded (which is typically done using a `CALL` and `POP` combination), you can obtain the address of the GOT, and you can then load the addresses of your variables out of linker−generated entries in the GOT.

The *data* section of a PIC shared library does not have these restrictions: since the data section is writable, it has to be copied into memory anyway rather than just paged in from the library file, so as long as it's being copied it can be relocated too. So you can put ordinary types of relocation in the data section without too much worry (but see section 8.2.4 for a caveat).

### 8.2.1 Obtaining the Address of the GOT

Each code module in your shared library should define the GOT as an external symbol:

```
extern  _GLOBAL_OFFSET_TABLE_   ; in ELF
extern  __GLOBAL_OFFSET_TABLE_  ; in BSD a.out
```

At the beginning of any function in your shared library which plans to access your data or BSS sections, you must first calculate the address of the GOT. This is typically done by writing the function in this form:

```
func:   push    ebp
        mov     ebp,esp
        push    ebx
        call    .get_GOT
.get_GOT:
        pop     ebx
        add     ebx,_GLOBAL_OFFSET_TABLE_+$$-.get_GOT wrt ..gotpc

        ; the function body comes here

        mov     ebx,[ebp-4]
        mov     esp,ebp
        pop     ebp
        ret
```

(For BSD, again, the symbol `_GLOBAL_OFFSET_TABLE` requires a second leading underscore.)

The first two lines of this function are simply the standard C prologue to set up a stack frame, and the last three lines are standard C function epilogue. The third line, and the fourth to last line, save and restore the `EBX` register, because PIC shared libraries use this register to store the address of the GOT.

The interesting bit is the `CALL` instruction and the following two lines. The `CALL` and `POP` combination obtains the address of the label `.get_GOT`, without having to know in advance where the program was loaded (since the `CALL` instruction is encoded relative to the current position). The `ADD` instruction makes use of one of the special PIC relocation types: GOTPC relocation. With the `WRT ..gotpc` qualifier specified, the symbol referenced (here `_GLOBAL_OFFSET_TABLE_`, the special symbol assigned to the GOT) is given as an offset from the beginning of the section. (Actually, `ELF` encodes it as the offset from the operand field of the `ADD` instruction, but NASM simplifies this deliberately, so you do things the same way for both `ELF` and `BSD`.) So the instruction then *adds* the beginning of the section, to get the real address of the GOT, and subtracts the value of `.get_GOT` which it knows is in `EBX`. Therefore, by the time that instruction has finished, `EBX` contains the address of the GOT.

If you didn't follow that, don't worry: it's never necessary to obtain the address of the GOT by any other means, so you can put those three instructions into a macro and safely ignore them:

```
%macro  get_GOT 0

        call    %%getgot
  %%getgot:
        pop     ebx
        add     ebx,_GLOBAL_OFFSET_TABLE_+$$-%%getgot wrt ..gotpc

%endmacro
```

### 8.2.2 Finding Your Local Data Items

Having got the GOT, you can then use it to obtain the addresses of your data items. Most variables will reside in the sections you have declared; they can be accessed using the `..gotoff` special `WRT` type. The way this works is like this:

```
        lea     eax,[ebx+myvar wrt ..gotoff]
```

The expression `myvar wrt ..gotoff` is calculated, when the shared library is linked, to be the offset to the local variable `myvar` from the beginning of the GOT. Therefore, adding it to `EBX` as above will place the real address of `myvar` in `EAX`.

If you declare variables as `GLOBAL` without specifying a size for them, they are shared between code modules in the library, but do not get exported from the library to the program that loaded it. They will still be in your ordinary data and BSS sections, so you can access them in the same way as local variables, using the above `..gotoff` mechanism.

Note that due to a peculiarity of the way BSD `a.out` format handles this relocation type, there must be at least one non−local symbol in the same section as the address you're trying to access.

### 8.2.3 Finding External and Common Data Items

If your library needs to get at an external variable (external to the *library*, not just to one of the modules within it), you must use the `..got` type to get at it. The `..got` type, instead of giving you the offset from the GOT base to the variable, gives you the offset from the GOT base to a GOT *entry* containing the address of the variable. The linker will set up this GOT entry when it builds the library, and the dynamic linker will place the correct address in it at load time. So to obtain the address of an external variable `extvar` in `EAX`, you would code

```
        mov     eax,[ebx+extvar wrt ..got]
```

This loads the address of `extvar` out of an entry in the GOT. The linker, when it builds the shared library, collects together every relocation of type `..got`, and builds the GOT so as to ensure it has every necessary entry present.

Common variables must also be accessed in this way.

### 8.2.4 Exporting Symbols to the Library User

If you want to export symbols to the user of the library, you have to declare whether they are functions or data, and if they are data, you have to give the size of the data item. This is because the dynamic linker has to build procedure linkage table entries for any exported functions, and also moves exported data items away from the library's data section in which they were declared.

So to export a function to users of the library, you must use

```
global  func:function           ; declare it as a function

func:   push    ebp

        ; etc.
```

And to export a data item such as an array, you would have to code

```
global  array:data array.end−array      ; give the size too

array:  resd    128
.end:
```

Be careful: If you export a variable to the library user, by declaring it as `GLOBAL` and supplying a size, the variable will end up living in the data section of the main program, rather than in your library's data section, where you declared it. So you will have to access your own global variable with the `..got` mechanism rather than `..gotoff`, as if it were external (which, effectively, it has become).

Equally, if you need to store the address of an exported global in one of your data sections, you can't do it by means of the standard sort of code:

```
dataptr:        dd      global_data_item        ; WRONG
```

NASM will interpret this code as an ordinary relocation, in which `global_data_item` is merely an offset from the beginning of the `.data` section (or whatever); so this reference will end up pointing at your data section instead of at the exported global which resides elsewhere.

Instead of the above code, then, you must write

```
dataptr:        dd      global_data_item wrt ..sym
```

which makes use of the special `WRT` type `..sym` to instruct NASM to search the symbol table for a particular symbol at that address, rather than just relocating by section base.

Either method will work for functions: referring to one of your functions by means of

```
funcptr:        dd      my_function
```

will give the user the address of the code you wrote, whereas

```
funcptr:        dd      my_function wrt .sym
```

will give the address of the procedure linkage table for the function, which is where the calling program will *believe* the function lives. Either address is a valid way to call the function.

### 8.2.5 Calling Procedures Outside the Library

Calling procedures outside your shared library has to be done by means of a *procedure linkage table*, or PLT. The PLT is placed at a known offset from where the library is loaded, so the library code can make calls to the PLT in a position−independent way. Within the PLT there is code to jump to offsets contained in the GOT, so function calls to other shared libraries or to routines in the main program can be transparently passed off to their real destinations.

To call an external routine, you must use another special PIC relocation type, `WRT ..plt`. This is much easier than the GOT−based ones: you simply replace calls such as `CALL printf` with the PLT−relative version `CALL printf WRT ..plt`.

### 8.2.6 Generating the Library File

Having written some code modules and assembled them to `.o` files, you then generate your shared library with a command such as

```
ld −shared −o library.so module1.o module2.o       # for ELF
ld −Bshareable −o library.so module1.o module2.o   # for BSD
```

For ELF, if your shared library is going to reside in system directories such as `/usr/lib` or `/lib`, it is usually worth using the `−soname` flag to the linker, to store the final library file name, with a version number, into the library:

```
ld −shared −soname library.so.1 −o library.so.1.2 *.o
```

You would then copy `library.so.1.2` into the library directory, and create `library.so.1` as a symbolic link to it.

# Chapter 9: Mixing 16 and 32 Bit Code

This chapter tries to cover some of the issues, largely related to unusual forms of addressing and jump instructions, encountered when writing operating system code such as protected−mode initialisation routines, which require code that operates in mixed segment sizes, such as code in a 16−bit segment trying to modify data in a 32−bit one, or jumps between different−size segments.

## 9.1 Mixed−Size Jumps

The most common form of mixed−size instruction is the one used when writing a 32−bit OS: having done your setup in 16−bit mode, such as loading the kernel, you then have to boot it by switching into protected mode and jumping to the 32−bit kernel start address. In a fully 32−bit OS, this tends to be the *only* mixed−size instruction you need, since everything before it can be done in pure 16−bit code, and everything after it can be pure 32−bit.

This jump must specify a 48−bit far address, since the target segment is a 32−bit one. However, it must be assembled in a 16−bit segment, so just coding, for example,

```
        jmp     0x1234:0x56789ABC       ; wrong!
```

will not work, since the offset part of the address will be truncated to `0x9ABC` and the jump will be an ordinary 16−bit far one.

The Linux kernel setup code gets round the inability of `as86` to generate the required instruction by coding it manually, using `DB` instructions. NASM can go one better than that, by actually generating the right instruction itself. Here's how to do it right:

```
        jmp     dword 0x1234:0x56789ABC         ; right
```

The `DWORD` prefix (strictly speaking, it should come *after* the colon, since it is declaring the *offset* field to be a doubleword; but NASM will accept either form, since both are unambiguous) forces the offset part to be treated as far, in the assumption that you are deliberately writing a jump from a 16−bit segment to a 32−bit one.

You can do the reverse operation, jumping from a 32−bit segment to a 16−bit one, by means of the `WORD` prefix:

```
        jmp     word 0x8765:0x4321      ; 32 to 16 bit
```

If the `WORD` prefix is specified in 16−bit mode, or the `DWORD` prefix in 32−bit mode, they will be ignored, since each is explicitly forcing NASM into a mode it was in anyway.

## 9.2 Addressing Between Different−Size Segments

If your OS is mixed 16 and 32−bit, or if you are writing a DOS extender, you are likely to have to deal with some 16−bit segments and some 32−bit ones. At some point, you will probably end up writing code in a 16−bit segment which has to access data in a 32−bit segment, or vice versa.

If the data you are trying to access in a 32−bit segment lies within the first 64K of the segment, you may be able to get away with using an ordinary 16−bit addressing operation for the purpose; but sooner or later, you will want to do 32−bit addressing from 16−bit mode.

The easiest way to do this is to make sure you use a register for the address, since any effective address containing a 32−bit register is forced to be a 32−bit address. So you can do

```
        mov     eax,offset_into_32_bit_segment_specified_by_fs
        mov     dword [fs:eax],0x11223344
```

This is fine, but slightly cumbersome (since it wastes an instruction and a register) if you already know the precise offset you are aiming at. The x86 architecture does allow 32−bit effective addresses to specify nothing but a 4−byte offset, so why shouldn't NASM be able to generate the best instruction for the purpose?

It can. As in section 9.1, you need only prefix the address with the `DWORD` keyword, and it will be forced to be a 32−bit address:

```
        mov     dword [fs:dword my_offset],0x11223344
```

Also as in section 9.1, NASM is not fussy about whether the `DWORD` prefix comes before or after the segment override, so arguably a nicer−looking way to code the above instruction is

```
        mov     dword [dword fs:my_offset],0x11223344
```

Don't confuse the `DWORD` prefix *outside* the square brackets, which controls the size of the data stored at the address, with the one `inside` the square brackets which controls the length of the address itself. The two can quite easily be different:

```
        mov     word [dword 0x12345678],0x9ABC
```

This moves 16 bits of data to an address specified by a 32−bit offset.

You can also specify `WORD` or `DWORD` prefixes along with the `FAR` prefix to indirect far jumps or calls. For example:

```
        call    dword far [fs:word 0x4321]
```

This instruction contains an address specified by a 16−bit offset; it loads a 48−bit far pointer from that (16−bit segment and 32−bit offset), and calls that address.

## 9.3 Other Mixed−Size Instructions

The other way you might want to access data might be using the string instructions (`LODSx`, `STOSx` and so on) or the `XLATB` instruction. These instructions, since they take no parameters, might seem to have no easy way to make them perform 32−bit addressing when assembled in a 16−bit segment.

This is the purpose of NASM's `a16` and `a32` prefixes. If you are coding `LODSB` in a 16−bit segment but it is supposed to be accessing a string in a 32−bit segment, you should load the desired address into `ESI` and then code

```
        a32     lodsb
```

The prefix forces the addressing size to 32 bits, meaning that `LODSB` loads from `[DS:ESI]` instead of `[DS:SI]`. To access a string in a 16−bit segment when coding in a 32−bit one, the corresponding `a16` prefix can be used.

The `a16` and `a32` prefixes can be applied to any instruction in NASM's instruction table, but most of them can generate all the useful forms without them. The prefixes are necessary only for instructions with implicit addressing: `CMPSx`, `SCASx`, `LODSx`, `STOSx`, `MOVSx`, `INSx`, `OUTSx`, and `XLATB`. Also, the various push and pop instructions (`PUSHA` and `POPF` as well as the more usual `PUSH` and `POP`) can accept `a16` or `a32` prefixes to force a particular one of `SP` or `ESP` to be used as a stack pointer, in case the stack segment in use is a different size from the code segment.

`PUSH` and `POP`, when applied to segment registers in 32−bit mode, also have the slightly odd behaviour that they push and pop 4 bytes at a time, of which the top two are ignored and the bottom two give the value of the segment register being manipulated. To force the 16−bit behaviour of segment−register push and pop instructions, you can use the operand−size prefix `o16`:

```
        o16 push    ss
        o16 push    ds
```

This code saves a doubleword of stack space by fitting two segment registers into the space which would normally be consumed by pushing one.

(You can also use the `o32` prefix to force the 32−bit behaviour when in 16−bit mode, but this seems less useful.)

# Chapter 10: Writing 64−bit Code (Unix, Win64)

This chapter attempts to cover some of the common issues involved when writing 64−bit code, to run under Win64 or Unix. It covers how to write assembly code to interface with 64−bit C routines, and how to write position−independent code for shared libraries.

All 64−bit code uses a flat memory model, since segmentation is not available in 64−bit mode. The one exception is the `FS` and `GS` registers, which still add their bases.

Position independence in 64−bit mode is significantly simpler, since the processor supports `RIP`–relative addressing directly; see the `REL` keyword (section 3.3). On most 64−bit platforms, it is probably desirable to make that the default, using the directive `DEFAULT REL` (section 5.2).

64−bit programming is relatively similar to 32−bit programming, but of course pointers are 64 bits long; additionally, all existing platforms pass arguments in registers rather than on the stack. Furthermore, 64−bit platforms use SSE2 by default for floating point. Please see the ABI documentation for your platform.

64−bit platforms differ in the sizes of the fundamental datatypes, not just from 32−bit platforms but from each other. If a specific size data type is desired, it is probably best to use the types defined in the Standard C header `<inttypes.h>`.

In 64−bit mode, the default instruction size is still 32 bits. When loading a value into a 32−bit register (but not an 8− or 16−bit register), the upper 32 bits of the corresponding 64−bit register are set to zero.

## 10.1 Immediates and displacements in 64−bit mode

In 64−bit mode, immediates and displacements are generally only 32 bits wide. NASM will therefore truncate most displacements and immediates to 32 bits.

The only instruction which takes a full 64−bit immediate is:

```
MOV reg64,imm64
```

NASM will produce this instruction whenever the programmer uses `MOV` with an immediate into a 64−bit register. If this is not desirable, simply specify the equivalent 32−bit register, which will be automatically zero−extended by the processor, or specify the immediate as `DWORD`:

```
mov rax,foo             ; 64−bit immediate
mov rax,qword foo       ; (identical)
mov eax,foo             ; 32−bit immediate, zero−extended
mov rax,dword foo       ; 32−bit immediate, sign−extended
```

The length of these instructions are 10, 5 and 7 bytes, respectively.

The only instructions which take a full 64−bit \e{displacement} is loading or storing, using `MOV`, `AL`, `AX`, `EAX` or `RAX` (but no other registers) to an absolute 64−bit address. Since this is a relatively rarely used instruction (64−bit code generally uses relative addressing), the programmer has to explicitly declare the displacement size as `QWORD`:

```
default abs

mov eax,[foo]           ; 32−bit absolute disp, sign−extended
mov eax,[a32 foo]       ; 32−bit absolute disp, zero−extended
mov eax,[qword foo]     ; 64−bit absolute disp

default rel

mov eax,[foo]           ; 32−bit relative disp
mov eax,[a32 foo]       ; d:o, address truncated to 32 bits(!)
mov eax,[qword foo]     ; error
mov eax,[abs qword foo] ; 64−bit absolute disp
```

A sign−extended absolute displacement can access from –2 GB to +2 GB; a zero−extended absolute displacement can access from 0 to 4 GB.

FIXME: THIS IS NOT YET CORRECTLY IMPLEMENTED

## 10.2 Interfacing to 64−bit C Programs (Unix)

On Unix, the 64−bit ABI is defined by the document:

`http://www.x86−64.org/documentation/abi.pdf`

Although written for AT&T−syntax assembly, the concepts apply equally well for NASM−style assembly. What follows is a simplified summary.

The first six integer arguments (from the left) are passed in `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`, in that order. Additional integer arguments are passed on the stack. These registers, plus `RAX`, `R10` and `R11` are destroyed by function calls, and thus are available for use by the function without saving.

Integer return values are passed in `RAX` and `RDX`, in that order.

Floating point is done using SSE registers, except for `long double`. Floating−point arguments are passed in `XMM0` to `XMM7`; return is `XMM0` and `XMM1`. `long double` are passed on the stack, and returned in `ST(0)` and `ST(1)`.

All SSE and x87 registers are destroyed by function calls.

On 64−bit Unix, `long` is 64 bits.

## 10.3 Interfacing to 64−bit C Programs (Win64)

The Win64 ABI is described at:

`http://msdn2.microsoft.com/en−gb/library/ms794533.aspx`

What follows is a simplified summary.

The first four integer arguments are passwd in `RCX`, `RDX`, `R8` and `R9`, in that order. Additional integer arguments are passed on the stack. These registers, plus `RAX`, `R10` and `R11` are destroyed by function calls, and thus are available for use by the function without saving.

Integer return values are passed in `RAX` only.

Floating point is done using SSE registers, except for `long double`. Floating−point arguments are passed in `XMM0` to `XMM3`; return is `XMM0` only.

On Win64, `long` is 32 bits; `long long` or `_int64` is 64 bits.

# Chapter 11: Troubleshooting

This chapter describes some of the common problems that users have been known to encounter with NASM, and answers them. It also gives instructions for reporting bugs in NASM if you find a difficulty that isn't listed here.

## 11.1 Common Problems

### 11.1.1 NASM Generates Inefficient Code

We sometimes get 'bug' reports about NASM generating inefficient, or even 'wrong', code on instructions such as `ADD ESP,8`. This is a deliberate design feature, connected to predictability of output: NASM, on seeing `ADD ESP,8`, will generate the form of the instruction which leaves room for a 32−bit offset. You need to code `ADD ESP,BYTE 8` if you want the space−efficient form of the instruction. This isn't a bug, it's user error: if you prefer to have NASM produce the more efficient code automatically enable optimization with the `−On` option (see section 2.1.17).

### 11.1.2 My Jumps are Out of Range

Similarly, people complain that when they issue conditional jumps (which are `SHORT` by default) that try to jump too far, NASM reports 'short jump out of range' instead of making the jumps longer.

This, again, is partly a predictability issue, but in fact has a more practical reason as well. NASM has no means of being told what type of processor the code it is generating will be run on; so it cannot decide for itself that it should generate `Jcc NEAR` type instructions, because it doesn't know that it's working for a 386 or above. Alternatively, it could replace the out−of−range short `JNE` instruction with a very short `JE` instruction that jumps over a `JMP NEAR`; this is a sensible solution for processors below a 386, but hardly efficient on processors which have good branch prediction *and* could have used `JNE NEAR` instead. So, once again, it's up to the user, not the assembler, to decide what instructions should be generated. See section 2.1.17.

### 11.1.3 `ORG` Doesn't Work

People writing boot sector programs in the `bin` format often complain that `ORG` doesn't work the way they'd like: in order to place the `0xAA55` signature word at the end of a 512−byte boot sector, people who are used to MASM tend to code

```
ORG 0

; some boot sector code

ORG 510
DW 0xAA55
```

This is not the intended use of the `ORG` directive in NASM, and will not work. The correct way to solve this problem in NASM is to use the `TIMES` directive, like this:

```
ORG 0

; some boot sector code

TIMES 510−($−$$) DB 0
DW 0xAA55
```

The `TIMES` directive will insert exactly enough zero bytes into the output to move the assembly point up to 510. This method also has the advantage that if you accidentally fill your boot sector too full, NASM will catch the problem at assembly time and report it, so you won't end up with a boot sector that you have to disassemble to find out what's wrong with it.

### 11.1.4 `TIMES` Doesn't Work

The other common problem with the above code is people who write the `TIMES` line as

```
TIMES 510−$ DB 0
```

by reasoning that `$` should be a pure number, just like 510, so the difference between them is also a pure number and can happily be fed to `TIMES`.

NASM is a *modular* assembler: the various component parts are designed to be easily separable for re−use, so they don't exchange information unnecessarily. In consequence, the `bin` output format, even though it has been told by the `ORG` directive that the `.text` section should start at 0, does not pass that information back to the expression evaluator. So from the evaluator's point of view, `$` isn't a pure number: it's an offset from a section base. Therefore the difference between `$` and 510 is also not a pure number, but involves a section base. Values involving section bases cannot be passed as arguments to `TIMES`.

The solution, as in the previous section, is to code the `TIMES` line in the form

```
TIMES 510−($−$$) DB 0
```

in which `$` and `$$` are offsets from the same section base, and so their difference is a pure number. This will solve the problem and generate sensible code.

## 11.2 Bugs

We have never yet released a version of NASM with any *known* bugs. That doesn't usually stop there being plenty we didn't know about, though. Any that you find should be reported firstly via the `bugtracker` at `https://sourceforge.net/projects/nasm/` (click on "Bugs"), or if that fails then through one of the contacts in section 1.2.

Please read section 2.2 first, and don't report the bug if it's listed in there as a deliberate feature. (If you think the feature is badly thought out, feel free to send us reasons why you think it should be changed, but don't just send us mail saying 'This is a bug' if the documentation says we did it on purpose.) Then read section 11.1, and don't bother reporting the bug if it's listed there.

If you do report a bug, *please* give us all of the following information:

- What operating system you're running NASM under. DOS, Linux, NetBSD, Win16, Win32, VMS (I'd be impressed), whatever.
- If you're running NASM under DOS or Win32, tell us whether you've compiled your own executable from the DOS source archive, or whether you were using the standard distribution binaries out of the archive. If you were using a locally built executable, try to reproduce the problem using one of the standard binaries, as this will make it easier for us to reproduce your problem prior to fixing it.
- Which version of NASM you're using, and exactly how you invoked it. Give us the precise command line, and the contents of the `NASMENV` environment variable if any.
- Which versions of any supplementary programs you're using, and how you invoked them. If the problem only becomes visible at link time, tell us what linker you're using, what version of it you've got, and the exact linker command line. If the problem involves linking against object files generated by a compiler, tell us what compiler, what version, and what command line or

options you used. (If you're compiling in an IDE, please try to reproduce the problem with the command−line version of the compiler.)

- If at all possible, send us a NASM source file which exhibits the problem. If this causes copyright problems (e.g. you can only reproduce the bug in restricted−distribution code) then bear in mind the following two points: firstly, we guarantee that any source code sent to us for the purposes of debugging NASM will be used *only* for the purposes of debugging NASM, and that we will delete all our copies of it as soon as we have found and fixed the bug or bugs in question; and secondly, we would prefer *not* to be mailed large chunks of code anyway. The smaller the file, the better. A three−line sample file that does nothing useful *except* demonstrate the problem is much easier to work with than a fully fledged ten−thousand−line program. (Of course, some errors *do* only crop up in large files, so this may not be possible.)
- A description of what the problem actually *is*. 'It doesn't work' is *not* a helpful description! Please describe exactly what is happening that shouldn't be, or what isn't happening that should. Examples might be: 'NASM generates an error message saying Line 3 for an error that's actually on Line 5'; 'NASM generates an error message that I believe it shouldn't be generating at all'; 'NASM fails to generate an error message that I believe it *should* be generating'; 'the object file produced from this source code crashes my linker'; 'the ninth byte of the output file is 66 and I think it should be 77 instead'.
- If you believe the output file from NASM to be faulty, send it to us. That allows us to determine whether our own copy of NASM generates the same file, or whether the problem is related to portability issues between our development platforms and yours. We can handle binary files mailed to us as MIME attachments, uuencoded, and even BinHex. Alternatively, we may be able to provide an FTP site you can upload the suspect files to; but mailing them is easier for us.
- Any other information or data files that might be helpful. If, for example, the problem involves NASM failing to generate an object file while TASM can generate an equivalent file without trouble, then send us *both* object files, so we can see what TASM is doing differently from us.

# Appendix A: Ndisasm

The Netwide Disassembler, NDISASM

## A.1 Introduction

The Netwide Disassembler is a small companion program to the Netwide Assembler, NASM. It seemed a shame to have an x86 assembler, complete with a full instruction table, and not make as much use of it as possible, so here's a disassembler which shares the instruction table (and some other bits of code) with NASM.

The Netwide Disassembler does nothing except to produce disassemblies of *binary* source files. NDISASM does not have any understanding of object file formats, like `objdump`, and it will not understand `DOS .EXE` files like `debug` will. It just disassembles.

## A.2 Getting Started: Installation

See section 1.3 for installation instructions. NDISASM, like NASM, has a `man page` which you may want to put somewhere useful, if you are on a Unix system.

## A.3 Running NDISASM

To disassemble a file, you will typically use a command of the form

```
        ndisasm −b @\{16|32|64@\} filename
```

NDISASM can disassemble 16−, 32− or 64−bit code equally easily, provided of course that you remember to specify which it is to work with. If no `−b` switch is present, NDISASM works in 16−bit mode by default. The `−u` switch (for USE32) also invokes 32−bit mode.

Two more command line options are `−r` which reports the version number of NDISASM you are running, and `−h` which gives a short summary of command line options.

### A.3.1 COM Files: Specifying an Origin

To disassemble a `DOS .COM` file correctly, a disassembler must assume that the first instruction in the file is loaded at address `0x100`, rather than at zero. NDISASM, which assumes by default that any file you give it is loaded at zero, will therefore need to be informed of this.

The `−o` option allows you to declare a different origin for the file you are disassembling. Its argument may be expressed in any of the NASM numeric formats: decimal by default, if it begins with '`$`' or '`0x`' or ends in '`H`' it's `hex`, if it ends in '`Q`' it's `octal`, and if it ends in '`B`' it's `binary`.

Hence, to disassemble a `.COM` file:

```
        ndisasm −o100h filename.com
```

will do the trick.

### A.3.2 Code Following Data: Synchronisation

Suppose you are disassembling a file which contains some data which isn't machine code, and *then* contains some machine code. NDISASM will faithfully plough through the data section, producing machine instructions wherever it can (although most of them will look bizarre, and some may have unusual prefixes, e.g. ‘`FS OR AX,0x240A`’), and generating ‘DB’ instructions ever so often if it’s totally stumped. Then it will reach the code section.

Supposing NDISASM has just finished generating a strange machine instruction from part of the data section, and its file position is now one byte *before* the beginning of the code section. It’s entirely possible that another spurious instruction will get generated, starting with the final byte of the data section, and then the correct first instruction in the code section will not be seen because the starting point skipped over it. This isn’t really ideal.

To avoid this, you can specify a ‘`synchronisation`’ point, or indeed as many synchronisation points as you like (although NDISASM can only handle 8192 sync points internally). The definition of a sync point is this: NDISASM guarantees to hit sync points exactly during disassembly. If it is thinking about generating an instruction which would cause it to jump over a sync point, it will discard that instruction and output a ‘`db`’ instead. So it *will* start disassembly exactly from the sync point, and so you *will* see all the instructions in your code section.

Sync points are specified using the `−s` option: they are measured in terms of the program origin, not the file position. So if you want to synchronize after 32 bytes of a `.COM` file, you would have to do

```
ndisasm −o100h −s120h file.com
```

rather than

```
ndisasm −o100h −s20h file.com
```

As stated above, you can specify multiple sync markers if you need to, just by repeating the `−s` option.

### A.3.3 Mixed Code and Data: Automatic (Intelligent) Synchronisation

Suppose you are disassembling the boot sector of a `DOS` floppy (maybe it has a virus, and you need to understand the virus so that you know what kinds of damage it might have done you). Typically, this will contain a `JMP` instruction, then some data, then the rest of the code. So there is a very good chance of NDISASM being *misaligned* when the data ends and the code begins. Hence a sync point is needed.

On the other hand, why should you have to specify the sync point manually? What you’d do in order to find where the sync point would be, surely, would be to read the `JMP` instruction, and then to use its target address as a sync point. So can NDISASM do that for you?

The answer, of course, is yes: using either of the synonymous switches `−a` (for automatic sync) or `−i` (for intelligent sync) will enable `auto−sync` mode. Auto−sync mode automatically generates a sync point for any forward−referring PC−relative jump or call instruction that NDISASM encounters. (Since NDISASM is one−pass, if it encounters a PC−relative jump whose target has already been processed, there isn’t much it can do about it...)

Only PC−relative jumps are processed, since an absolute jump is either through a register (in which case NDISASM doesn’t know what the register contains) or involves a segment address (in which case the target code isn’t in the same segment that NDISASM is working in, and so the sync point can’t be placed anywhere useful).

For some kinds of file, this mechanism will automatically put sync points in all the right places, and save you from having to place any sync points manually. However, it should be stressed that auto−sync mode is *not* guaranteed to catch all the sync points, and you may still have to place some manually.

Auto−sync mode doesn’t prevent you from declaring manual sync points: it just adds automatically generated ones to the ones you provide. It’s perfectly feasible to specify `−i` *and* some `−s` options.

Another caveat with auto−sync mode is that if, by some unpleasant fluke, something in your data section should disassemble to a PC−relative call or jump instruction, NDISASM may obediently place a sync point in a totally random place, for example in the middle of one of the instructions in your code section. So you may end up with a wrong disassembly even if you use auto−sync. Again, there isn't much I can do about this. If you have problems, you'll have to use manual sync points, or use the `−k` option (documented below) to suppress disassembly of the data area.

### A.3.4 Other Options

The `−e` option skips a header on the file, by ignoring the first N bytes. This means that the header is *not* counted towards the disassembly offset: if you give `−e10 −o10`, disassembly will start at byte 10 in the file, and this will be given offset 10, not 20.

The `−k` option is provided with two comma−separated numeric arguments, the first of which is an assembly offset and the second is a number of bytes to skip. This *will* count the skipped bytes towards the assembly offset: its use is to suppress disassembly of a data section which wouldn't contain anything you wanted to see anyway.

## A.4 Bugs and Improvements

There are no known bugs. However, any you find, with patches if possible, should be sent to `nasm−bugs@lists.sourceforge.net`, or to the developer's site at `https://sourceforge.net/projects/nasm/` and we'll try to fix them. Feel free to send contributions and new features as well.

Future plans include awareness of which processors certain instructions will run on, and marking of instructions that are too advanced for some processor (or are `FPU` instructions, or are undocumented opcodes, or are privileged protected−mode instructions, or whatever).

That's All Folks!

I hope NDISASM is of some use to somebody. Including me. :−)

I don't recommend taking NDISASM apart to see how an efficient disassembler works, because as far as I know, it isn't an efficient one anyway. You have been warned.

# Index

- `!` operator, unary 28
- `!=` operator 43
- `$$` token 27, 70
- `$`
  - Here token 27
  - prefix 22, 25, 72
- `%` operator 28
- `%!` 54
- `%$` and `%$$` prefixes 46, 47
- `%%` operator 28, 37
- `%+` 34
- `%+1` and `%−1` syntax 41
- `%0` parameter count 39
- `&` operator 27
- `&&` operator 43
- `*` operator 28
- `+` modifier 38
- `+` operator
  - binary 28
  - unary 28
- `−` operator
  - binary 28
  - unary 28
- `..@` symbol prefix 31, 37
- `/` operator 28
- `//` operator 28
- `<` operator 43
- `<<` operator 28
- `<=` operator 43
- `<>` operator 43
- `=` operator 43
- `==` operator 43
- `>` operator 43
- `>=` operator 43
- `>>` operator 28
- `?` MASM syntax 23
- `^` operator 27
- `^^` operator 43
- `|` operator 27
- `||` operator 43
- `~` operator 28
- `−a` option 17, 103
- `a16` 95
- `a32` 95
- `a86` 10, 19, 20
- `ABS` 25
- `ABSOLUTE` 58, 65
- addition 28
- addressing, mixed−size 94
- address−size prefixes 22
- algebra 25
- `ALIGN` 51, 62, 64
- `ALIGNB` 51
- alignment
  - in `bin` sections 63
  - in `elf` sections 69
  - in `obj` sections 64
  - in `win32` sections 68
  - of `elf` common variables 71
- `ALINK` 75
- `alink.sourceforge.net` 75
- `alloc` 69
- `alt.lang.asm` 10, 11
- ambiguity 20
- `a.out`
  - BSD version 71
  - Linux version 71
- `aout` 13, 71
- `aoutb` 71, 90
- `%arg` 52
- `arg` 83, 89
- `as86` 10, 13, 72
- assembler directives 56
- assembly passes 29
- assembly−time options 16
- `%assign` 35
- `ASSUME` 20
- `AT` 51
- Autoconf 11
- `autoexec.bat` 11
- `auto-sync` 103
- `−b` 102
- bin 13, 14, 62
  - multisection 63
- binary 25
- binary files 23
- 16−bit mode, versus 32−bit mode 56
- 64−bit \e{displacement 97
- 64−bit immediate 97
- bit shift 28
- `BITS` 56, 62
- `__BITS__` 50
- bitwise AND 27
- bitwise OR 27

bitwise XOR 27
block IFs 47
boot loader 62
boot sector 99
Borland
  Pascal 84
  Win32 compilers 63
braces
  after `%` sign 40
  around macro parameters 36
BSD 90
`.bss` 69, 71, 72
bugs 100
`bugtracker` 100
`BYTE` 99
C calling convention 80, 87
C symbol names 78
`CALL FAR` 29
case sensitivity 19, 32, 33, 35, 36, 43, 65
changing sections 57
character constant 23, 26
circular references 32
`CLASS` 64
`%clear` 49
`c16.mac` 83, 86
`c32.mac` 89
`coff` 13, 69
colon 22
`.COM` 62, 77
command−line 13, 62
commas in macro parameters 38
`COMMON` 60, 64
  `elf` extensions to 71
  `obj` extensions to 67
Common Object File Format 69
common variables 60
  alignment in `elf` 71
  element size 67
`comp.lang.asm.x86` 10, 11
`comp.os.linux.announce` 11
`comp.os.msdos.programmer` 78
concatenating macro parameters 40
condition codes as macro parameters 41
conditional assembly 41
conditional jumps 99
conditional−return macro 41
`configure` 11
constants 25
context stack 46, 47
context−local labels 46
context−local single−line macros 47
counting macro parameters 39
`CPU` 60
`CPUID` 26
creating contexts 46
critical expression 23, 24, 29, 35, 58
`−D` option 16
`−d` option 16
`.data` 69, 71, 72
`_DATA` 80
`data` 70, 73
data structure 82, 89
`DB` 23, 26
`dbg` 73
`DD` 23, 26
debug information 15
debug information format 14
declaring structures 50
`DEFAULT` 57
`default` 71
default macro parameters 38
default name 62
default−`WRT` mechanism 67
`%define` 16, 32
defining sections 57
design goals 19
DevPac 23, 31
disabling listing expansion 41
division 28
DJGPP 69, 87
`djlink` 75
DLL symbols
  exporting 66
  importing 66
`DO` 23, 26
DOS 11, 15
DOS archive 11
DOS source archive 11
`DQ` 23, 26
`.drectve` 68
`DT` 23, 26
`DUP` 21, 24
`DW` 23, 26
`DWORD` 23
`−E` option 17
`−e` option 17, 104
effective addresses 22, 24, 30
element size, in common variables 67
ELF 13, 69
  shared libraries 70
  16−bit code and 71
`elf, elf32, and elf64` 69
`%elif` 42, 43
`%elifctx` 43
`%elifdef` 42
`%elifid` 44

`%elifidn` 43
`%elifidni` 43
`%elifmacro` 42
`%elifn` 43
`%elifnctx` 43
`%elifndef` 42
`%elifnid` 44
`%elifnidn` 43
`%elifnidni` 43
`%elifnmacro` 42
`%elifnnum` 44
`%elifnstr` 44
`%elifnum` 44
`%elifstr` 44
`%else` 42
e−mail 11
`endproc` 83, 89
`%endrep` 45
`ENDSTRUC` 50, 58
environment 19
`EQU` 23, 24, 30
`%error` 44
error messages 15
error reporting format 15
`EVEN` 51
`.EXE` 63, 75
`EXE_begin` 76
`EXE2BIN` 78
`exebin.mac` 76
`exec` 69
Executable and Linkable Format 69
`EXE_end` 76
`EXE_stack` 76
`%exitrep` 45
`EXPORT` 66
`export` 73
exporting symbols 59
expressions 17, 27
extension 13, 62
`EXTERN` 59
    `obj` extensions to 67
    `rdf` extensions to 73
`−F` option 14
`−f` option 14, 62
far call 20
far common variables 67
far pointer 29
`FARCODE` 83, 86
`__FILE__` 49
`FLAT` 65
flat memory model 87
flat−form binary 62
`FLOAT` 61
`__FLOAT__` 61
`__float16__` 26
`__float32__` 26
`__float64__` 26
`__FLOAT_DAZ__` 61
`__float80e__` 26
`__float128h__` 26
floating−point 20, 22, 23, 26
    constants 26, 61
`__float128l__` 26
`__float80m__` 26
`__FLOAT_ROUND__` 61
`follows=` 63
format−specific directives 56
forward references 30
frame pointer 80, 84, 87
FreeBSD 71, 90
FreeLink 75
`ftp.kernel.org` 11
`ftp.simtel.net` 75
`function` 70, 73
functions
    C calling convention 80, 87
    Pascal calling convention 84
`−g` option 15
`gas` 10
`gcc` 10
`GLOBAL` 59
    `aoutb` extensions to 70
    `elf` extensions to 70
    `rdf` extensions to 73
global offset table 90
`_GLOBAL_OFFSET_TABLE_` 70
`gnu−elf−extensions` 18
`..got` 70
`GOT` relocations 92
GOT 70, 90
`..gotoff` 70
`GOTOFF` relocations 91
`..gotpc` 70
`GOTPC` relocations 91
graphics 23
greedy macro parameters 37
`GROUP` 65
groups 28
`−h` 102
hex 25
`hidden` 71
hybrid syntaxes 20
`−I` option 15
`−i` option 15, 103
`%iassign` 35
`ibiblio.org` 11

`%idefine` 32
`IEND` 51
`%if` 41, 43
`%ifctx` 42, 47
`%ifdef` 42
`%ifid` 43
`%ifidn` 43
`%ifidni` 43
`ifmacro` 42
`%ifn` 43
`%ifnctx` 43
`%ifndef` 42
`%ifnid` 44
`%ifnidn` 43
`%ifnidni` 43
`%ifnmacro` 42
`%ifnnum` 44
`%ifnstr` 44
`%ifnum` 43
`%ifstr` 43
`%imacro` 36
`IMPORT` 66
import library 66
importing symbols 59
`INCBIN` 23, 26
`incbin` 15
`%include` 15, 16, 45
include search path 16
including other files 45
inefficient code 99
infinite loop 27
`__Infinity__` 27
infinity 27
informational section 68
`INSTALL` 12
installing 11
instances of structures 51
Intel number formats 27
`internal` 71
`ISTRUC` 51
iterating over macro parameters 39
`Jcc NEAR` 99
`JMP DWORD` 94
jumps, mixed−size 94
`−k` 104
`−l` option 14
label prefix 31
`ld86` 72
`LIBRARY` 72
license 10
`%line` 54
`__LINE__` 49
linker, free 75
Linux
  `a.out` 71
  `as86` 72
  ELF 69
listing file 14
little−endian 26
`%local` 53
local labels 30
logical AND 43
logical negation 28
logical OR 43
logical XOR 43
`−M` option 14
mac osx 69
mach object file format 69
`macho` 13, 69
`%macro` 36
macro library 16
macro processor 32
macro−local labels 37
`macro−params` 18
macros 24
`macro−selfref` 18
`make` 11
makefile dependencies 14
makefiles 11
`Makefile.unx` 12
man pages 11
map files 63
`MASM` 10
MASM 19, 24, 63
memory models 20, 79
memory operand 23
memory references 19, 24
`−MG` option 14
Microsoft OMF 63
Minix 72
`misc` subdirectory 76, 83, 89
mixed−language program 78
mixed−size addressing 94
mixed−size instruction 94
MMX registers
ModR/M byte
`MODULE` 72
modulo operators 28
MS−DOS 62
MS−DOS device drivers 78
multi−line macros 18, 36
multipass optimization 17
multiple section names 62
multiplication 28
`multipush` macro 39
`Multisection` 63

__NaN__ 27
NaN 27
nasm.1 11
NASM version 49
nasm version id 49
nasm version string 49
__NASMDEFSEG 64
nasm−devel 11
nasm.exe 11
nasm −f <format> −y 14
nasm −hf 14
__NASM_MAJOR__ 49
__NASM_MINOR__ 49
nasm.out 13
___NASM_PATCHLEVEL__ 49
__NASM_SUBMINOR__ 49
__NASM_VER__ 49
__NASM_VERSION_ID__ 49
nasmw.exe 11
nasmXXXs.zip 11
nasm−X.XX.tar.gz 11
nasmXXX.zip 11
ndisasm.1 11
ndisasm 102
ndisasm.exe 11
ndisasmw.exe 11
near call 20
near common variables 67
NetBSD 71, 90
new releases 11
noalloc 69
nobits 63, 69
noexec 69
.nolist 41
‘nowait’ 21
nowrite 69
number−overflow 18
numeric constants 23, 25
−o option 13, 102
o16 96
o32 96
.OBJ 75
obj 13, 63
object 70, 73
octal 25
OF_DBG 73
OF_DEFAULT 14
OFFSET 20
OMF 63
omitted parameters 38
−On option 17
one’s complement 28
OpenBSD 71, 90
operands 22
operand−size prefixes 22
operating system 62
    writing 94
operators 27
ORG 62, 77, 78, 99
orphan−labels 18, 22
OS/2 63, 65
other preprocessor directives 54
out of range, jumps 99
output file format 14
output formats 62
overlapping segments 28
OVERLAY 65
overloading
    multi−line macros 36
    single−line macros 33
−P option 16
−p option 16, 46
paradox 29
PASCAL 86
Pascal calling convention 84
passes, assembly 29
PATH 11
period 30
Perl 11
perverse 16
PharLap 64
PIC 70, 72, 90
..plt 70
PLT relocations 70, 92, 93
plt relocations 93
%pop 46
position−independent code 70, 72, 90
--postfix 19
precedence 27
pre−defining macros 16, 33
preferred 28
--prefix 19
pre−including files 16
preprocess−only mode 17
preprocessor 17, 24, 28, 32
preprocessor expressions 17
preprocessor loops 45
preprocessor variables 35
primitive directives 56
PRIVATE 64
proc 73, 83, 89
procedure linkage table 70, 92, 93
processor mode 56
progbits 63, 69
program entry point 66, 75
program origin 62

`protected` 71
pseudo−instructions 23
`PUBLIC` 59, 64
pure binary 62
`%push` 46
`__QNaN__` 27
quick start 19
`QWORD` 23
`−r` 102
`rdf` 13, 72
`rdoff` subdirectory 12, 72
redirecting errors 15
`REL` 25, 57
relational operators 43
Relocatable Dynamic Object File Format 72
relocations, PIC−specific 70
removing contexts 46
renaming contexts 47
`%rep` 24, 45
repeating 24, 45
`%repl` 47
reporting bugs 100
`RESB` 21, 23, 29
`RESD` 23
`RESO` 23
`RESQ` 23
`REST` 23
`RESW` 23
`%rotate` 39
rotating macro parameters 39
`−s` option 15, 103
searching for include files 46
`__SECT__` 57, 58
`SECTION` 57
    `elf` extensions to 69
    `win32` extensions to 68
section alignment
    in `bin` 63
    in `elf` 69
    in `obj` 64
    in `win32` 68
section, bin extensions to 62
`SEG` 28, 64
`SEGMENT` 57
    `elf` extensions to 64
segment address 28
segment alignment
    in `bin` 63
    in `obj` 64
segment names, Borland Pascal 85
segment override 20, 22
segments 28
    groups of 65
separator character 19
shared libraries 72, 90
shared library 70
`shift` command 39
SIB byte
signed division 28
signed modulo 28
single−line macros 32
size, of symbols 70
`__SNaN__` 27
Solaris x86 69
`−soname` 93
sound 23
source code 11
source−listing file 14
square brackets 19, 24
`STACK` 64
`%stacksize` 53
standard macros 49
standardized section names 57, 68, 69, 71, 72
`..start` 66, 75
`start=` 63
`stderr` 15
`stdout` 15
`STRICT` 29
string constant 23
string handling in macros 35
string length 35
`%strlen` 35
`STRUC` 50, 58, 82, 89
stub preprocessor 17
`%substr` 35
sub−strings 35
subtraction 28
suppressible warning 18
suppressing preprocessing 17
switching between sections 57
`..sym` 70
symbol sizes, specifying 70
symbol types, specifying 70
symbols
    exporting from DLLs 66
    importing from DLLs 66
`synchronisation` 103
`.SYS` 62, 78
`−t` 17
`TASM` 10, 17
tasm 19, 63
tasm compatible preprocessor directives 52

`TBYTE` 21
`test` subdirectory 75
testing
  arbitrary numeric expressions 43
  context stack 42
  exact text identity 43
  multi−line macro existence 42
  single−line macro existence 42
  token types 43
`.text` 69, 71, 72
`_TEXT` 80
`TIMES` 23, 24, 29, 99, 100
`TLINK` 78
trailing colon 22
two−pass assembler 29
`TWORD` 21, 23
type, of symbols 70
`−U` option 16
`−u` option 16, 102
unary operators 28
`%undef` 16, 34
undefining macros 16
underscore, in C symbols 78
uninitialized 23
uninitialized storage 21
Unix 11
  SCO 69
  source archive 11
  System V 69
UnixWare 69
unrolled loops 24
unsigned division 28
unsigned modulo 28
`UPPERCASE` 19, 65
`USE16` 57, 65
`USE32` 57, 65
user−defined errors 44
user−level assembler directives 49
user−level directives 56
`−v` option 18
VAL 75
valid characters 22
variable types 20
version 18
version number of NASM 49
`vfollows=` 63
Visual C++ 68
`vstart=` 63
`−w` option 18
warnings 18
`[warning +warning−name]` 18
`[warning −warning−name]` 18
`win64` 69, 97
Win64 11, 13, 63, 68, 87
Windows 75
Windows 95 11
Windows NT 11
`write` 69
writing operating systems 94
`WRT` 28, 64, 70, 72
`WRT ..got` 92
`WRT ..gotoff` 91
`WRT ..gotpc` 91
`WRT ..plt` 93
`WRT ..sym` 93
WWW page 10
`www.cpan.org` 11
`www.delorie.com` 75
`www.pcorner.com` 75
`−X` option 15
`%xdefine` 33
`x2ftp.oulu.fi` 75
`%xidefine` 33
`−y` option 19
`−Z` option 15